Programming CG Tools

SDK Version 1.0

Contents

Code Examples

Equations

Figures

Tables

1 C3 library

1.1 Overview

The C3 library exports data from a CG tool to the NINTENDO GAMECUBE (GCN) runtime format. CG tool programmers will find this document helpful in explaining how to use or port the C3 library to various CG tools.

1.1.1 Architecture

During development, we used 3D Studio MAX primarily, so our converter for this CG tool in the NINTENDO GAMECUBE Character Pipeline (CP) SDK is more mature. We've also included an export tool for Maya 3.0, but this one is less robust. In any case, the C3 library was coded with maximum portability in mind, so other CG tool data conversion paths may be constructed quickly. Figure 1 shows the 3D Studio MAX conversion path, which is similar to that of Maya 3.0.

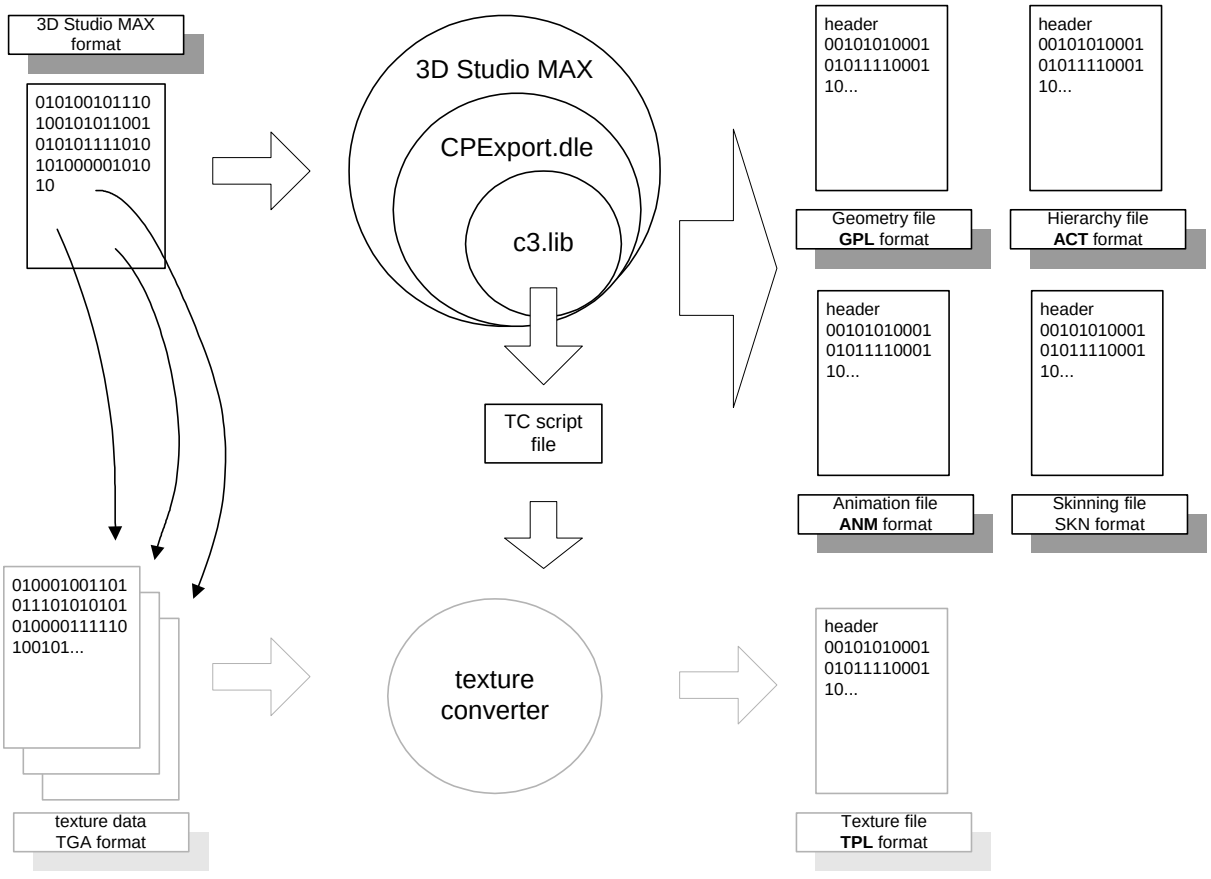

Figure 1 3D Studio MAX conversion path

To maximize portability, we've divided the conversion process into several modules:

- The export plug-in, in this case `CPExport.dle`, the 3D Studio MAX Release 3.1 converter.
- 3D conversion library (`c3.lib`).
- Texture conversion path.

`CPExport.dle` is entirely 3D Studio MAX-specific; i.e., it uses MAX functions and C++ class definitions to access MAX data structures. Respectively, `CPExport.mll` is a converter specific to Maya 3.0. Both tools extract data that is then loaded into the C3 library, which provides APIs to perform the following operations:

- Extract data from the CG tool into C3 internal data structures.
- Optimize data for GCN hardware.

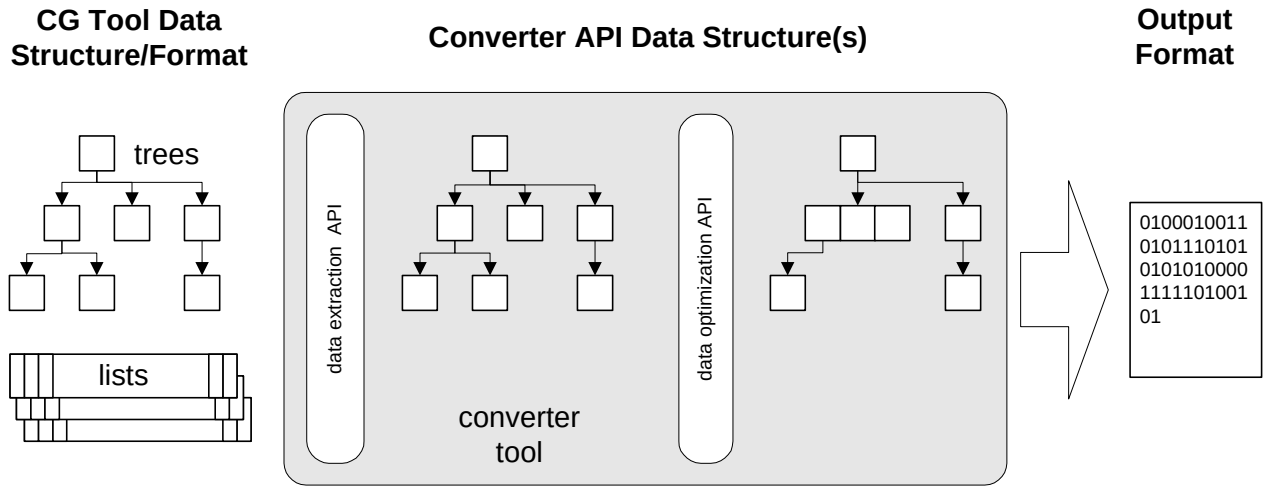

Figure 2 C3 library architecture

In order to support a particular CG tool, therefore, you need only to understand the data extraction API and format supplied by the CG tool vendor, and the data optimization API.

To learn more about the texture conversion path, please refer to Chapter 2. To learn more about how to visually preview the data on the GCN hardware or emulator, see Chapter 1 in "Character Pipeline for Artists."

1.1.2 Source code

In order to view the header and source files, please use Visual C++ 6.0 to load the desired workspace:

- C3 library: `/cp/build/libraries/c3/vc++/c3.dsw`.
- 3D Studio MAX converter: `/cp/build/tools/Max/CPExport/vc++/MaxConv.dsw`.
- Maya converter: `/cp/build/tools/Maya/CPExport/vc++/MayaConv.dsw`.

1.1.3 Program flow

In order to utilize the C3 library properly, you must follow a simple program flow order:

1. Initialization.
2. Option setting.

3. Extraction.
4. Optimization.
5. Output.
6. Conversion clean-up.

The following code segment from `CPExport.dle` uses the C3 library and exemplifies this progression:

```
// Initialize C3 library
C3Initialize();

// Send options using C3 options API to control data optimization and extraction
SetOptions( &maxOptions );

// Extract data from 3DS MAX SDK using C3 extraction API
C3ReportStatus( "Extracting information from Max..." );
iNode = gi->GetRootNode();
numChild = iNode->NumberOfChildren();
for( int i = 0; i < numChild; i++ )          // Traverse the tree of geometry objects
{
    ProcessINode( iNode->GetChildNode(i) );  // 3DS MAX function using C3 extraction API
}

// Optimize the data: compress duplicates, weld, and remove null primitives.
// Finish processing optimization options: compute quantization shift bits
// Optimize geometry: convert triangles to strips and fans
// Optimize actor: convert vertex stitching info, prune bones
C3OptimizeBeforeOutput();

// Write the files: assemble display list, quantize, and output to GPL, ACT, ANM
// Write TCS texture conversion script and then call TexConv to generate TPL
C3WriteFile( mPath, mFile );

// Cleanup C3 library
C3Clean();
```

Code 1 Conversion program flow

These sets of functions are all that a CG tool programmer needs, except for those routines to set the options and extract data that should be written individually. The API to set options is detailed in section 1.2.2. The extraction API is explained separately in the relevant sections for geometry, texture, hierarchy, and animation.

If you are interested in learning about the internal architecture and program flow of the C3 library, please refer to Appendix E.

1.1.4 Output

The C3 library can output up to six files:

- Geometry data (GPL).
- Hierarchy/bones data, also known as actor data (ACT).
- Animation data (ANM).
- Texture conversion script (TCS) used by `TexConv` to generate texture data (TPL).
- Skinning data for runtime CPU skinning (SKN).
- Statistics of geometry data.

The 3D Studio MAX CPExport code also exports a camera and light setup file (STP) specifically for the GCN previewer. The C3 library does not export the STP format since cameras and lights are game-specific. For more information, refer to “Character Pipeline for Artists” in this guide.

In the following sections, we will examine in detail the geometry (GPL), hierarchy (ACT), and animation (ANM) formats, as well as the conversion process.

1.2 Geometry

1.2.1 Functionality

The C3 library can load the following primitives and attributes:

- Triangles, quads, and lines.
- Vertex attributes: position, texture coordinate, color with alpha, normal, and weights for stitching/CPU skinning.
- Objects with multiple textures. (Note: We do not mean multiple textures per polygon; rather, we mean objects with different textures for different groups of polygons).
- Texture attributes: wrap/clamp property and filter method (point sample, bilerp, trilerp/mipmap).

The C3 library can perform the following optimizations:

- Compression by removing duplicate vertex data; C3 can also weld position and texture coordinates.
- Triangle strip and fan generation to maximize hardware performance; C3 can enable a strip and fan view to set vertex colors of primitives based on type.
- Quantization of vertex attributes to 8 bits or 16 bits, signed or unsigned.
- Display list creation with indexed position, texture coordinate, color, and normal (C3 uses indexed methods to refer to all vertex attributes), with minimal index byte size selection for display list.
- Indexing of normals into a default normal table of 252 normals.
- Primitive list sorting to minimize state changes among display lists.
- Output of all GCN texture formats.

1.2.2 Options

Options control the C3 library behavior during extraction, optimization, and output. The option API functions are all prefixed with "`C3SetOption`" or "`C3GetOption`"; we have shown only those functions which set an option:

```
// Feedback options
void C3SetOptionReportStatusFunc         ( C3ReportStatusFunc func );
void C3SetOptionReportErrorFunc          ( C3ReportErrorFunc func );

// Output options
void C3SetOptionFileExportFlag           ( u32 fileExportflag );
void C3SetOptionOutputEndian             ( u8 endianType );

// Geometry general options
void C3SetOptionSrcVertexOrder           ( u8 vtxOrder );
void C3SetOptionEnableStitching          ( C3Bool flag );
void C3SetOptionEnableLighting           ( C3Bool flag );
void C3SetOptionAmbientPercentage        ( f32 percent );

// Geometry optimization options
void C3SetOptionCompress                 ( u16 targets );
void C3SetOptionWeldRadius               ( u32 target, f32 radius );
void C3SetOptionEnableStripFan           ( C3Bool flag );
void C3SetOptionEnableStripFanView       ( C3Bool flag );
void C3SetOptionPositionRange            ( f32 range );
void C3SetOptionQuantization             ( u32 target, u8 channel, u8 quantInfo );
void C3SetOptionUseDefaultNormalTable    ( C3Bool flag );
void C3SetOptionUseExternalNormalTable   ( C3Bool flag );
```



```
void C3SetOptionExternalNormalTablePath  ( char* name );
```

Code 2 C3 option-setting API

You can understand how to use this option-setting API properly by referring to section E.2.2 in the Appendix. Becoming familiar with how to use the GCN converter will also help you to get a feel for the effects these options have. Section 1.2.4 explains the geometry optimization options in more detail.

1.2.3 Extraction

Since the 3D data we need to extract from a CG tool is hierarchical, we have designed the C3 API to extract geometry data into its internal C3 representation hierarchically. The code segment below shows how functions may be nested:

```
C3BeginObject
    C3SetColor                      // color for entire object

    C3BeginTexture                  // texture for entire object
        C3SetTexFmt
        C3SetImage
        C3SetImageAlpha
        C3SetTexTiling
        C3SetTexFilterType
        C3SetImgLOD
    C3EndTexture

    C3BeginLinePrimitive            // line primitives (can do line strips)
        C3SetColor                  // color for primitive, override object color
        C3BeginVertex
            C3SetPosition
            C3SetColor              // color per vertex, override primitive/object color
            C3SetNormal
            C3SetWeight
        C3EndVertex
    C3EndLinePrimitive

    C3BeginPolyPrimitive            // polygon primitives (only triangles. quads are untested)
        C3SetColor                  // color for primitive, override object color
        C3BeginTexture              // texture per primitive, override object texture
            ...
        C3EndTexture
        C3BeginVertex
            C3SetPosition
            C3SetTextureCoord
            C3SetColor              // color per vertex, override primitive/object color
            C3SetNormal
            C3SetWeight
        C3EndVertex
    C3EndPolyPrimitive
C3EndObject
```

Code 3 Nested extraction functions

These functions and data all have stack-based states, so if color and texture are defined after `C3BeginObject`, the same color and texture will continue in all primitives and all vertices until primitives or vertices override the current state.

1.2.3.1 Loading vertex data

You can load vertex data easily by using the following functions between `C3BeginVertex` and `C3EndVertex`:

Function	Description
`C3SetPosition(float x, float y, float z)`	Sets the position. This function must be called.
`C3SetTexCoord(float s, float t, u8 chnl)`	Sets the texture coordinate for the given texture channel. Currently, the only texture channel supported is 0.
`C3SetColor(u8 r, u8 g, u8 b, u8 a)`	Sets the vertex color, including alpha.
`C3SetNormal(float x, float y, float z)`	Sets the normal.
`C3SetWeight(char* boneName, f32 weight)`	Sets one bone to which this vertex is attached, as well as the amount of its influence. The name must match exactly with bone names sent to `C3BeginHierarchyNode(char* name)`.

Table 1 API for loading vertex data

It is not necessary to call all of these functions. For example, if you do not wish to export normals, do not call `C3SetNormal` for any of the current geometry object's vertices.

Use the `C3SetWeight` function to assign weights to vertices for use in skinning or stitching. If all of the vertices in a geometry object have a weight of 1.0, the object will be converted as a stitched object; otherwise, the C3 library will output an additional SKN file to be used by the Character Pipeline runtime CPU skinning library.

1.2.3.2 Loading primitive data

C3 can load two types of primitives:

- **Lines:** You can create lines and line strips by calling `C3BeginLinePrimitive` and `C3EndLinePrimitive` between `C3BeginObject` and `C3EndObject`, and loading data for at least two vertices.
- **Polygons:** Code 4 illustrates how all calls must be between the `C3BeginObject` and `C3EndObject` functions in order to load polygons. This example shows how to load a triangle:


```
C3BeginObject( “SimpleTri” );
C3BeginPolyPrimitive();

C3BeginVertex();
C3SetPosition( 0, 0, 0 );
C3SetColor( 255, 0, 0, 255 );
C3EndVertex();

C3BeginVertex();
C3SetPosition( 20, 20, 0 );
C3SetColor( 0, 0, 255, 255 );
C3EndVertex();

C3BeginVertex();
C3SetPosition( 0, 20, 0 );
C3SetColor( 0, 255, 0, 255 );
C3EndVertex();

C3EndPolyPrimitive();
C3EndObject();
```

Code 4 SimpleTri.c

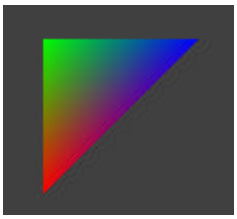

Figure 3 SimpleTri

You can run this segment after setting options, then follow it with calls to the rest of the functions outlined in section 1.1.2, to produce the triangle in Figure 3. You can also use the C3 library to load quads.

1.2.4 Optimization

Optimization in the C3 library is driven by the options set previously (see section 1.2.2).

1.2.4.1 Compression and welding

Call `C3SetOptionCompress` to compress positions, texture coordinates, colors, and normals independently in order to remove duplicate data and save space.

Since CG tools do not allow precise control in arranging positions or texture coordinates, small errors may occur. Welding can eliminate these errors by collapsing all data within a given radius to the same data. Since welding is a superset of compression through the removal of duplicate vertex data, welding can further reduce the size of the output data as well as maximize triangle stripping performance. Enable welding by calling `C3SetOptionWeldRadius` with a non-zero argument.

1.2.4.2 Triangle strip and fan

You can enable a simple stripping algorithm by calling `C3SetOptionEnableStripFan`:

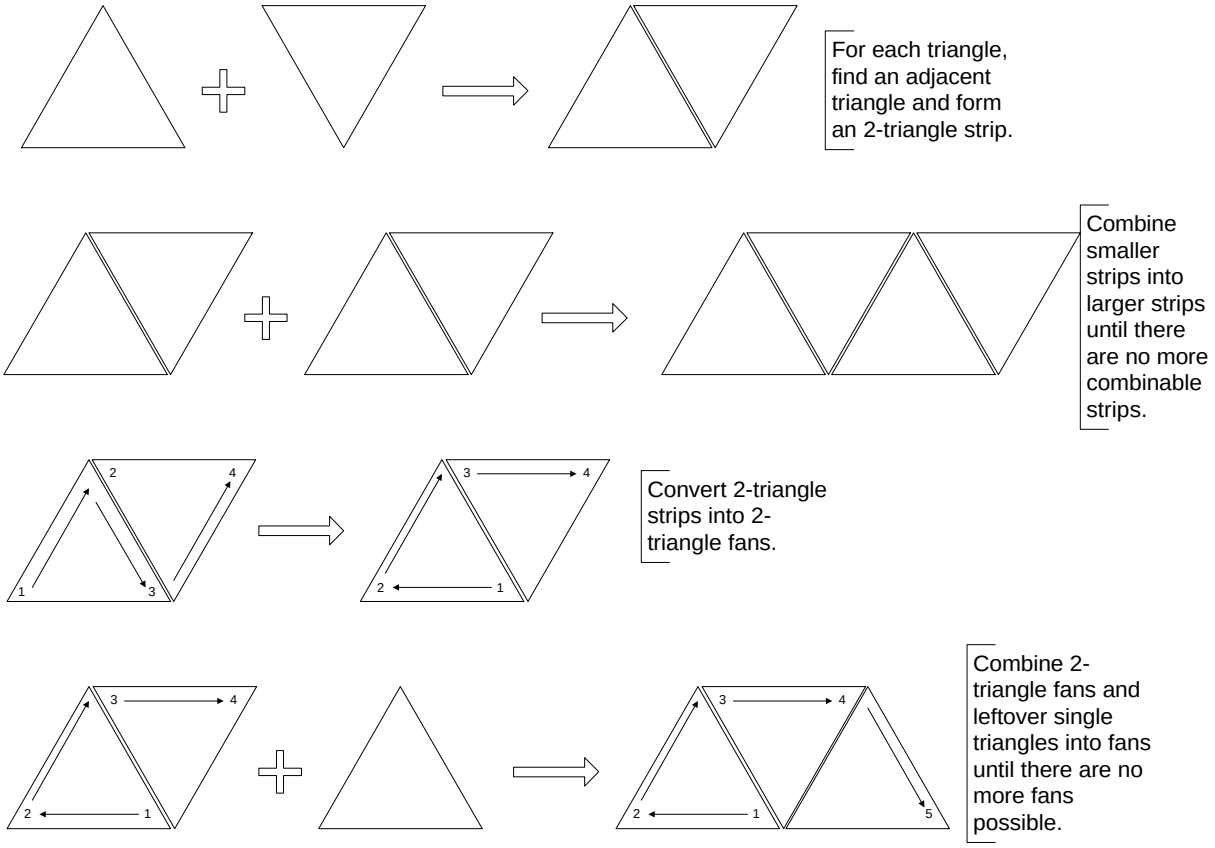

Figure 4 C3 Triangle strip and fan algorithm

There are two methods of evaluating the performance of the C3 triangle strip and fan algorithm. The first is by the number of vertices per triangle given in the C3 statistics file. The worst case is three vertices per triangle, meaning that no strips, fans, or quads were created. If all primitives were converted into quads, then there would be two vertices per triangle; one vertex per triangle is the best theoretical limit. The lower the number, the better the triangle strip/fan generation performance.

The second method is through visual feedback enabled with `C3SetOptionEnableStripFanView`. C3 can set vertex colors for primitives based on the primitive type: red for triangles, blue for quads, and random colors for triangle strips and fans. C3 will also export a wireframe of the triangles so that strips and fans can be distinguished by their triangle edges. Problem areas may thus be spotted and fixed.

By definition, triangle strips and fans in C3 break when any of the vertex attributes are not considered to be the same. We therefore recommend use of the C3 welding feature to minimize slight errors in position and texture coordinates that may be caused by CG tool or floating point imprecision. Please also remember that a triangle strip or fan can have only one texture.

Due to limitations in the current development hardware, there is a maximum limit of four vertices in any triangle fan converted in C3; however, this restriction should not be an issue in subsequent hardware releases.

You should also be aware of the preprocessor flag `C3_QUADS_TO_FANS` in `C3Defines.h`. This flag converts all quads to fans to work around a Macintosh OpenGL bug which allows non-coplanar quads to be wholly clipped if any one of the triangles are backfacing. This flag should be disabled for the Dolphin Development Hardware (DDH), but it should be enabled when running the GCN previewer on a PowerMac.

1.2.4.3 Quantization

`C3SetOptionQuantization` sets the quantization type to output positions, colors, texture coordinates, and normals as 32-bit floating point numbers or fixed point numbers. When outputting to fixed point, the number of necessary fractional bits (shift bits) is computed a little differently among positions, texture coordinates, and normals.

1.2.4.3.1 Position quantization

C3 can output positions to all GX formats:

- U8 8-bit unsigned.
- S8 8-bit signed.
- U16 16-bit unsigned.
- S16 16-bit signed.
- FLOAT 32-bit float.

Since it is desirable to quantize all non-skinned positions to a global fixed grid, you should specify a position range when quantizing positions to fixed point. Otherwise, if the position range is 0, C3 will automatically calculate the optimal position range. C3 will reserve only as many integer bits in the fixed point number as necessary to store the position range; the remaining bits are used for fractions.

Please be careful to call `C3SetOptionPositionRange` before setting position quantization type with `C3SetOptionQuantization(C3_TARGET_POSITION, …)`.

A fixed global grid of positions in the CG tool can be created among various geometry objects. Since positions are quantized according to a local coordinate system set by `C3SetHierControl` (see section 1.2.3), local transforms should have the same scale and should be rotationally orthogonal. We've included a simple example in `/cp/max/test/PosQnt` which tests how all positions can be quantized to a fixed grid.

Due to optimizations in the runtime CPU skinning library, skinned positions and normals will always be exported to a signed 16-bit format.

1.2.4.3.2 Vertex color quantization

C3 can output vertex colors to all GX formats:

- 32-bit RGBA (8888) and RGBXA (888X).
- 24-bit RGBA (6666) and RGB (888).
- 16-bit RGBA (4444) and RGB (565).

C3 is optimized to export vertex colors per geometry object with or without the alpha component, depending on whether it uses transparent vertex alpha or not. A minor test case in `/cp/max/test/ColorQnt` shows that for the given model, there is not much color quality trade-off between exporting colors in 24-bit RGB8 or 16-bit RGB565.

We expect that color may be compressed more efficiently by generating CLUTs. Compression quality will likely depend heavily upon pre-assigned vertex lighting. Currently, this technique is not implemented in the C3 library.

1.2.4.3.3 Texture coordinate quantization

C3 can output texture coordinates to all GX formats:

- U8 8-bit unsigned.
- S8 8-bit signed.
- U16 16-bit unsigned.
- S16 16-bit signed.
- FLOAT 32-bit float.

Texture coordinates are quantized in the same manner as positions, except that C3 individually calculates the number of integer bits necessary to store the maximum texture coordinate value for each object. The remaining bits of the fixed point number are used for fractions.

You may also consider generating lookup tables for texture coordinates, however, we have not tried this technique and do not have any information regarding the trade-off between compression and quality.

1.2.4.3.4 Normal quantization

C3 can output positions to all GX formats:

- U8 8-bit unsigned.
- S8 8-bit signed.
- U16 16-bit unsigned.
- S16 16-bit signed.
- FLOAT 32-bit float.

When C3 quantizes normals to a fixed point number format, it reserves one bit for the integer portion since the values of all normals are always between –1 and 1, inclusive. The remaining bits are used for the fractional portion of the fixed point number.

C3 can utilize normal tables; in this case, C3 will not quantize and output normal data since the normal table will be loaded at runtime. Normal indices into the normal table will still be provided in the display list.

NOTE: The user cannot control the type of quantization for normals in a skinned object, since the runtime CPU skinning library requires normals to be quantized to S16.

1.2.4.3.5 Keyframe quantization

The C3 library implements basic animation compression by quantizing all keyframe information into one of the GCN fixed point formats (see above). Quaternions and ease information within the keyframe is always quantized to S16 (1.14) because their values are always between –1 and 1, inclusive.

1.2.4.4 Minimal index selection

C3 automatically selects an 8-bit or 16-bit indexed array for positions, texture coordinates, colors, and normals depending on the number of elements in each array (e.g., if an array has fewer than 256 elements, C3 selects an 8-bit index instead of a 16-bit index). The test case in `/cp/max/test/IndexQnt` tests index selection as well as comparing output size differences.

1.2.4.5 Normal table

C3 can index into a global normal table for all geometry objects instead of exporting a normal array for each geometry object, which is a very good method of saving space. A default normal table of 252 normals which point roughly uniformly in all directions is provided in `normalTable.c`. The test case `/cp/max/test/NrmTabGn` shows how a normal table can be generated easily by using a special define called `C3_GENERATE_NORMAL_TABLE` (found in `C3Defines.h`).

1.2.4.6 Primitive sorting

C3 sorts primitives to minimize state changes when creating display lists. For example, some of the important state changes that can occur in stitched primitives are texture changes and matrix loads. This sorting occurs automatically in the C3 library; for more information, please consult `/cp/max/test/DlistSrt`.

1.3 Texture

1.3.1 Functionality

The C3 library can set any texture as long as it is supported by `TexConv`, described in further detail in section 2.1.2.

1.3.2 Extraction

C3 can load textures for a whole geometry object or per primitive, depending on whether the call to `C3BeginTexture`/`C3EndTexture` comes between `C3BeginObject`/`C3EndObject` or `C3BeginPolyPrimitive`/`C3EndPolyPrimitive`, respectively.

You can call the following functions between `C3BeginTexture` and `C3EndTexture`:

Function	Description
`C3SetTexFmt(C3TexFmt texFmt)`	Sets the current texture format for a color map, alpha map, or palettes.
`C3SetImage(char* fileName)`	Sets the relevant color map from the given file.
`C3SetImageAlpha(char* fileName)`	Sets the relevant alpha map from the given file.
`C3SetPalImage(char* fileName)`	Sets the relevant paletted image from the given file.
`C3SetTexTiling(u8 wrap)`	Sets whether the texture should be wrapped or clamped.
`C3SetTexFilterType(u8 method)`	Sets whether the texture should be point-sampled, bilerped, or mipmapped.
`C3SetImgLOD(u8 minLOD, u8 maxLOD, u8 baseLOD)`	Sets the number of mipmap LODs.

Table 2 C3 texture extraction API

Texture formats and wrap and filter mode constants are outlined in `C3Texture.h`. A test case using all of the Character Pipeline-supported texture formats is located in `/cp/max/test/TexFrmts`.

Using some of the preceding routines, the `SimpleTri` example can be extended by adding a texture:

```
C3BeginObject( “TexturedTri” );
C3BeginPolyPrimitive();

C3SetColor( 255, 255, 255, 255 );

C3BeginTexture( 0 );
C3SetTexFmt( RGB565 );
C3SetImage( “D:\\temp\\srcImage\\wood2.tga” );
C3SetTexTiling( C3_REPEAT_S | C3_REPEAT_T );
C3SetTexFilterType( C3_FILTER_LINEAR );
C3EndTexture();

C3BeginVertex();
C3SetPosition( 0, 0, 0 );
C3SetTexCoord( 0, 0, 0 );
C3EndVertex();

C3BeginVertex();
C3SetPosition( 20, 20, 0 );
C3SetTexCoord( 1, 1, 0 );
C3EndVertex();

C3BeginVertex();
C3SetPosition( 0, 20, 0 );
C3SetTexCoord( 0, 1, 0 );
C3EndVertex();

C3EndPolyPrimitive();
C3EndObject();
```

Code 5 TexturedTri.c

Figure 5 TexturedTri

Currently, the C3 library supports only one texture channel, so the argument to `C3BeginTexture`, as well as the last argument to `C3SetTexCoord`, must always be 0.

1.4 Actor

1.4.1 Functionality

The C3 library provides three basic services for hierarchies:

- Specifies relationships between geometry objects in a hierarchy.
- Performs instancing.
- Orders display priority.

There is no support for non-inheritance.

1.4.2 Extraction

The extraction of hierarchy information is simple using the C3 API:

```
void   C3BeginHierarchyNode ( char* name );
void   C3SetHierControl     ( u8 controlType, f32 x, f32 y, f32 z, f32 w );
                                                    // sets the transformation from parent bone
void   C3SetPivotOffset     ( f32 x, f32 y, f32 z ); // offsets the rotation and scale pivot
void   C3SetObject          ( char* identifier );    // links a geometry object to this bone
void   C3SetObjectSkin      ( char* identifier );    // links a stitched object to this root bone
void   C3SetDisplayPriority ( u8 priority );         // assigns a display order priority
void   C3EndHierarchyNode   ( );
```

Code 6 C3 hierarchy extraction API

C3 supports the extraction of hierarchy information in one of two ways:

- *Recursively*, by nesting the function calls to `C3BeginHierarchyNode`, or
- *Iteratively*, by using `C3SetParent` to set parenting information manually.

The function `C3SetHierControl` sets translation, rotation, and scale controls (specified by the *controlType*) relative to the parent transformation.

NOTE: The C3 library no longer supports arbitrary hierarchy matrices with the obsolete `C3SetMatrix`. In some transformations, the rotation and scale pivot point may not necessarily be in the same place as the translation from the parent; therefore, `C3SetPivotOffset` provides a method to specify the position offset of the rotation and scale pivot from the translation. The latter function may be necessary in Maya, but is not necessary for 3D Studio MAX.

In order to associate a specific geometry object to the current hierarchy node, call `C3SetObject` with the exact name of the object. If the geometry object is stitched, then call `C3SetObjectSkin`. Currently, only one stitched object is allowed per export.

Referring to the `SimpleTri` and `TexturedTri` examples, the following code segment shows how to arrange them, respectively, in a parent-child relationship. This code can be called before or after the objects are extracted.


```
C3BeginHierarchyNode( “Parent” );
// C3SetHierControl not called so identity assumed
C3SetObject( “SimpleTri” );

C3BeginHierarchyNode( “Child” );
C3SetHierControl( CTRL_TRANS, 40, 0, 0, 0 );
C3SetObject( “TexturedTri” );
// No need to use C3SetParent since parenting
// is assumed by the nesting of C3BeginHierarchyNode
C3EndHierarchyNode();

C3EndHierarchyNode();
```

Code 7 Creating hierarchy

Figure 6 Two triangles in hierarchy

1.4.2.1 Instancing

Since the hierarchy simply indexes geometry objects, you can perform instancing simply by using `C3SetObject` again with the name of a previously extracted object. Then, `C3SetHierControl` can be used to position, rotate, or scale the geometry object uniquely.

1.4.2.2 Display order priority

Since the Character Pipeline does not support multitexturing, we have used duplicate layers of geometry with vertex alpha to simulate it on the GCN previewer for Macintosh. (Note, however, that the same effect can be achieved in GCN hardware with one layer of geometry—making display priority unnecessary.)

Objects should be depth-sorted at runtime to determine proper display order. Runtime depth sorting is not currently supported, however, so objects may be sorted before runtime to minimize the improper display of transparent objects. This works particularly well for terrain that uses vertex alpha in conjunction with multiple layers of geometry (such as the `Knoll` database), since the camera will always be above the terrain.

All objects have a default display priority of 0, unless `C3SetDisplayPriority` is called between `C3BeginObject` and `C3EndObject`. All non-transparent objects should use the default display priority, while transparent objects should use an appropriate display priority between 1 and 254 (set by calling `C3SetDisplayPriority`).

Geometry objects in a scene will be rendered in ascending display priority in the GCN previewer. For example, if an object “Box” has a display priority of 10, and an object “Transparent Sphere” has a display priority of 20, then “Transparent Sphere” will always be drawn after “Box.”

Objects with the same display priority will be drawn in some arbitrary order.

1.4.3 Optimization

The C3 library performs automatic pruning of unused hierarchy bones to save space. A bone will be pruned if all of the following conditions exist:

- There is no attached geometry object.
- There is no attached vertices in the case of stitching.
- The first two conditions are true for all of the bone’s children.

As you can see from the conditions above, bone pruning works properly only if geometry is extracted, or else all bones will be pruned. Since the C3 library allows selective output of geometry and hierarchy, we have chosen to prune bones only if geometry is output. If geometry is not output, no pruning will occur.

NOTE: Outputting geometry means that `C3_FILE_GEOMETRY` is a set flag in `C3GetOptionFileExportFlag`, not that geometry is extracted with `C3BeginObject`/`C3EndObject`. In other words, geometry can still be *extracted* without being *output*, so bones will not be pruned.

1.5 Animation

1.5.1 Functionality

The C3 library can only extract keyframe animation of hierarchy bones. Rotations may be specified using quaternions or Euler XYZ angles in degrees. Major types of keyframe interpolation methods are supported, including linear, Bezier, and TCB (tension, continuity, bias) with ease.

There are several restrictions when extracting and converting animation information:

- All scaling in the hierarchy transformation should be positive.
- Keyframes must exist at the beginning and end of the animation range.
- All keyframes in a track must contain the same combination of position, rotation, and scale information. For example, if a track is animating position and rotation without scale, then all keyframes in that track must contain only position and rotation information.

The C3 library has only been tested to extract one sequence of animation per export because 3D Studio MAX allows only one animation per file. However, the ANM format has the capability to store multiple animation sequences. To support this, we have written a separate application called `AnmCombine` to splice ANM files together into one ANM file. For more information, please consult Appendix D.

1.5.2 Extraction

The C3 API for animation extraction continues to use the hierarchical `begin/end` paradigm:

```
void   C3BeginAnimation           ( char* hierNodeName );
void   C3EndAnimation             ( );

void   C3BeginTrack               ( char* sequenceName );
void   C3SetStartTime             ( float time );
void   C3SetEndTime               ( float time );
void   C3SetInterpTypeTranslation ( u8 interpType );
void   C3SetInterpTypeScale       ( u8 interpType );
void   C3SetInterpTypeRotation    ( u8 interpType );
void   C3EndTrack                 ( );

void   C3BeginKeyFrame            ( float time );
void   C3SetKeyTranslation        ( float x, float y, float z );
void   C3SetKeyScale              ( float x, float y, float z );
void   C3SetKeyRotationQuat       ( float x, float y, float z, float w );
void   C3SetKeyRotationEuler      ( float x, float y, float z );
void   C3SetKeyMatrix             ( MtxPtr mtx );
void   C3SetKeyInControl          ( u8 controlType, float x, float y, float z, float w );
void   C3SetKeyOutControl         ( u8 controlType, float x, float y, float z, float w );
void   C3SetKeyTCB                ( u8 controlType, float tension, float continuity, float bias );
void   C3SetKeyEase               ( u8 controlType, float easeIn, float easeOut );
```



```
void    C3EndKeyFrame              ( );
```

Code 8 C3 animation extraction API

First, here is a definition of terms:

- A keyframe describes the position, rotation, and scale for a bone at a certain time.
- A track is a collection of keyframes.
- A sequence is a collection of tracks.

Note that `C3BeginAnimation` cannot nest itself recursively (unlike `C3BeginHierarchyNode`), and `C3EndAnimation` must be called before another animation can begin. If `C3BeginAnimation` is called between `C3BeginHierarchyNode` and `C3EndHierarchyNode` to animate that particular hierarchy node, then a `NULL` argument should be sent. Otherwise, `C3BeginAnimation` should be called with the name of the hierarchy node it will animate.

1.5.2.1 Tracks

Call `C3BeginTrack`/`C3EndTrack` between `C3BeginAnimation` and `C3EndAnimation`. As tracks are created with new sequence names, the new sequences are created automatically. If no sequence name is provided, tracks will be added to a default sequence called "`NULL`". Set start and end times using `C3SetStartTime` and `C3SetEndTime`.

The interpolation type must be specified for the keyframes in this track using the `C3SetInterpType*` functions before keyframes are extracted. The interpolation types possible are listed in `C3AnmExt.h`. Note that the C3 library has no concept of how keyframes should be looped within a track.

1.5.2.2 Keyframes

Call `C3BeginKeyFrame`/`C3EndKeyFrame` between `C3BeginTrack` and `C3EndTrack`.

The transformation information in keyframes can be either a matrix (which cannot be interpolated) or some combination of translation, rotation, and scale. You may need to provide additional information depending on the interpolation type of the track.

Interpolation Type	Transformation Information
Linear, Slerp, Squad	No interpolation information needed.
Bezier	In and Out tangent angle in radians should be specified using `C3SetKeyInControl` and `C3SetKeyOutControl`.
Hermite (TCB) or Squad with ease in/out (SquadEE)	Tension, continuity, and bias parameters should be specified using `C3SetKeyTCB`, and ease in/out parameters should be set using `C3SetKeyEase`. In and Out tangent angles will be computed by the C3 library appropriately.

Table 3 Keyframe transformations

The following code segment shows how to animate any bone 90 degrees about the bone's *y*-axis:

```
C3BeginAnimation();

C3BeginTrack( “Rotate90Y” );
C3SetStartTime( 0 );
C3SetEndTime( 100 );
C3SetInterpTypeRotation( C3_INTERPTYPE_SQUAD );

C3BeginKeyFrame( 0 );
C3SetKeyRotationQuat( 0, 0, 0, 1 );
C3EndKeyFrame();

C3BeginKeyFrame( 50 );
C3SetKeyRotationQuat( 0, 0.7071f, 0, 0.7071f ); // 90 degrees around Y axis
C3EndKeyFrame();

C3BeginKeyFrame( 100 );
C3SetKeyRotationQuat( 0, 0, 0, 1 );
C3EndKeyFrame();

C3EndTrack();

C3EndAnimation();
```

Code 9 Bone animation

1.5.3 Optimization

Currently, there is no significant optimization in the Character Pipeline, except for the quantization of keyframe data (described in section 1.2.4.3.5).

2 TC library

2.1 Overview

- The Texture Converter (TC) library converts a collection of artist-generated textures into the GCN Texture Palette (TPL) format.
- The conversion "recipe" is specified in a texture conversion script file (TCS) that includes the file names, mipmap information, output formats, and TPL packing order.
- Plug-in architecture allows tool programmers to write file readers for any input file type.
- `TexConv.exe` is a texture conversion application that acts as a wrapper for the TC library. It includes a simple user interface and a default TGA file reading function.

2.1.1 Texture data pipeline

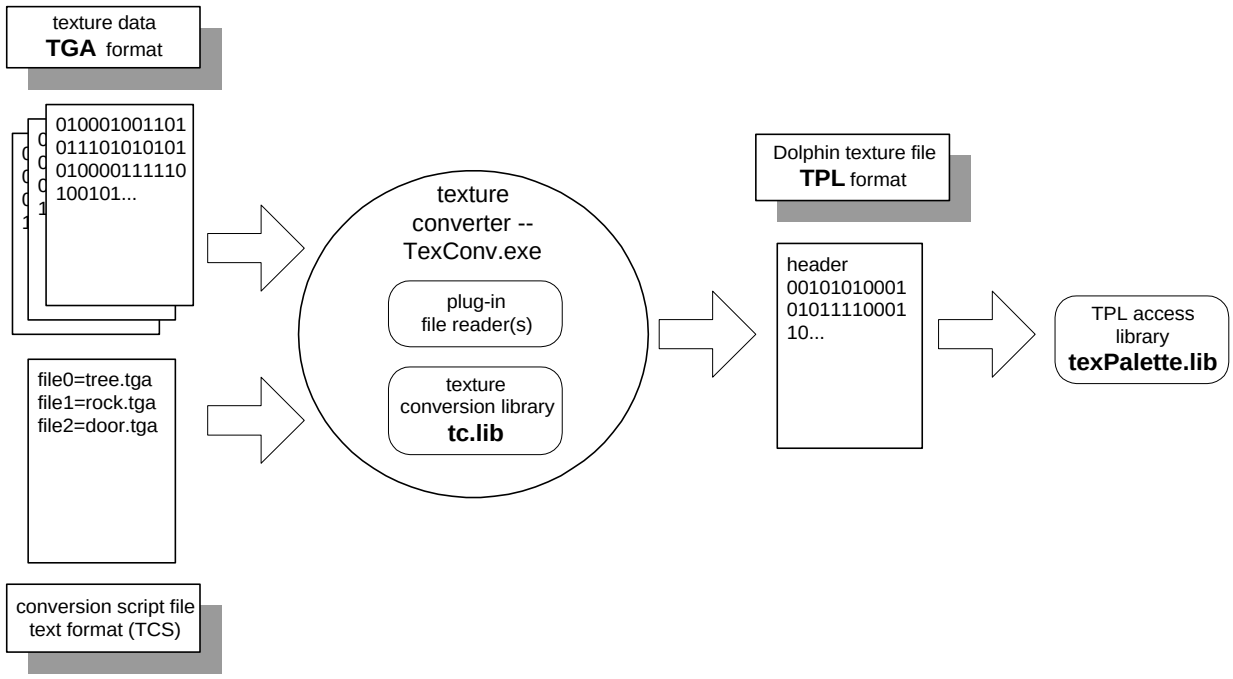

Figure 7 Texture data pipeline

The TPL file format packs multiple textures into a single file in order to speed disk loading time. For the sake of efficiency, TPL files should be composed of textures that share temporal locality within the game application. TPL files may contain many different texture maps and color lookup tables in different formats.

At runtime, games can use functions supplied by the texPalette library to access any texture in the TPL file and send it directly to the GX library.

2.1.2 Functionality

The TC library has many features, the most important of which include:

- Conversion to all non-Z GCN texture formats including RGBA, intensity alpha (IA), intensity (I), color-indexed (CI), and compressed (CMPR). All texture data in TPL files is 32-byte tiled and can be used by the GX library directly.
- Conversion of input palettes to RGB565 or RGB5A3 output format.
- Mipmap generation.
- S3TC texture compression (see Appendix B).
- Automatic color conversion (e.g., RGBA to IA, CI to RGB, RGB to RGBA)

NOTE: The GX library refers to a color palette as a Texture Lookup Table (TLUT), while many other tools call it a Color Lookup Table (CLUT).

2.1.2.1 Optimizations

We assume that artists will want to make many small changes to their TGA files before they settle on a final look. It should therefore not be necessary to incur the overhead of regenerating an entire TPL file each time an artist makes a small change to a texture.

To increase conversion speed, TC has the ability to update only those parts of a TPL file which have changed since the last execution. When the texture converter runs, it compares each new image and palette to the contents of the most recently-generated TPL file. If any data blocks match, the pre-existing blocks are copied directly to the new TPL file, bypassing the conversion process.

For a complete explanation of partial conversion rules, see section 2.4.2.

2.1.2.2 Color indexed texture support

Compression and animation are the two main reasons to use color-indexed textures. However, because the CMPR (S3TC) format offers high quality compressed textures from true color images, we expect many people to use this type of texture compression instead of CI. Therefore, we have not designed TC to support CLUT generation from true color images. This should not be confused with TC's ability to convert automatically from a CI input format to an RGB output format.

Moreover, TC does not support any CI texture animation encoding rules because it is difficult for us to define how to "encode" the CLUT for CI texture animation. These encoding rules are often specific to the desired animation, and the animation techniques are often specific to the game in question. TC support for CI textures is therefore very simple, our intent being only to show developers how to convert CI textures and CLUTs for The GCN to use, not necessarily the best way to generate them for a specific game.

Internally, TC stores color-indexed textures in 16-bit format. Output indices are copied directly from input indices. There is no shifting or masking of bits. Therefore, it is up to the programmer to ensure that the input color indices lie entirely within the output bit range. For example, if output is CI4, input indices should range from 0-15.

Internally, TC stores palette entries in RGBA format. Palettes may be output in either RGB565 or RGB5A3 formats. IA8 is not supported for palette entries.

NOTE: The 3D Studio MAX converter assumes that each TGA color indexed texture file will contain both the CI texture map and CLUT. 3D Studio MAX then instructs `TexConv` to use this CI texture map with this CLUT. Please note that the 3D Studio MAX converter does not provide any method for sharing an input CLUT with multiple input CI texture maps.

2.1.3 Program flow

We provide the full source code for both the `TexConv` program and the TC library. To assist code comprehension, this section provides an overview of the major processing steps TC takes to convert raw TGA files into a TPL file. If you intend to alter the source code, read this section first.

Figure 8 shows the logical flow for making a TPL file with the TC library. Here are the major steps involved:

1. **Input:** In this phase of the program, the texture converter reads and parses the input script file. Error checking is performed to make sure that all the data is coherent and that all requested input files exist. If these requirements are not met, the program exits with an error message.

2. **Setup:** `TexConv` gathers together the input data and organizes it for conversion. During this phase `TexConv` creates and manages four lists of information:

   - *Source Image List* – Lists all the raw input files along with some information about size and format.
   - *Image List* – Contains information about how different raw files are combined to form a final image. For instance, an image can take its RGB information from one source and get its alpha information from another.
   - *Palette List* – Maintains palette information from source data. Palettes are extracted from the source files and stored here.
   - *Texture List* – Contains binding information to associate images with palettes in the output TPL file.

3. **Conversion/Output:** During the conversion/output phase, `TexConv` reads through the texture list and converts the raw data into GCN format data. If pre-converted data exists, it is used directly and the conversion operation is skipped. During conversion, TC generates mipmap levels as necessary. Once a texture is fully converted and processed, it is output to the new TPL file. When all the textures in the texture list have been processed, the program closes the TPL file, which is now ready for the runtime libraries.

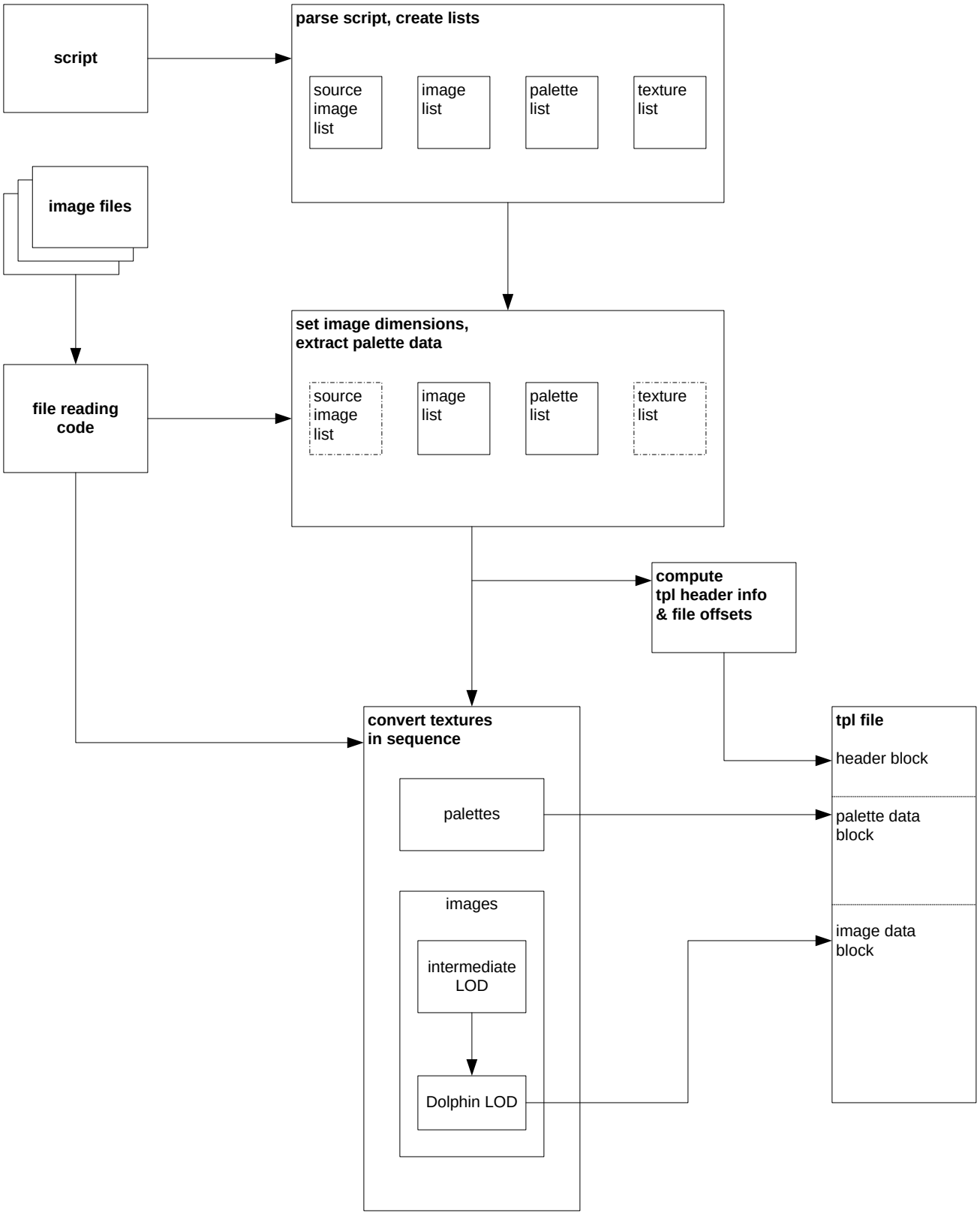

Figure 8 TC library program flow

2.1.4 Source code

Header and source files for `TexConv.exe` and the TC library can be viewed in Visual C++ 6.0 by loading

`/cp/build/tools/TexConv/vc++/TexConv.dsw`.

2.2 Input

2.2.1 Source image files

The `TexConv` program has a single plug-in file reader. We chose to support the True Graphics Adapter (TGA) format since it is capable of storing many different texture formats, including true color, intensity, intensity alpha, color indexed, and 256 color palettes. The TGA format has been available for years; many graphics tools can import and export it.

For a detailed description of the TGA format, please refer to James Murray, et al., *Encyclopedia of Graphics File Formats, 2nd Ed.*, O'Reilly & Assoc., Sebastopol, CA, 1996, pp. 860-879.

To learn how to add more plug-in file readers, please refer to section 4.5.

2.2.2 Script file

The following is a sample texture converter script file:

```
                                    ; comments begin with a semicolon and continue to
                                    ; the end of a line

path = c:/level1/mario/             ; path sets directory for files
                                    ; if path = NULL, full path is required for file names
                                    ; path is pre-concatenated to all subsequent file names
                                    ; path may be set on any line
                                    ; path may be absolute or relative

file    0  =  marioHead_rgba8.tga   ; full file name is c:/level1/mario/ marioHead_rgba8.tga
image   0  =  0, 0, RGBA8           ; image0[RGB]=file0,image0[A]=file0, convert to RGBA8
texture 0  =  0, x                  ; texture 0 of TPL file. Image index is image0
                                    ; CLUT index is 'x' since it's not a color-indexed texture

path       =  d:/temp/mario/        ; change path
file    1  =  marioArm_rgb565.tga   ; full file name is d:/temp/mario/marioArm_rgb565.tga
image   1  =  1, x, RGB565, 0, 3, 0 ; generate mipmap 0 through 3 (4 LODs) and remap to 0
texture 1  =  1, x

file    2  =  marioFoot_ci8.tga
image   2  =  2, x, CI8             ; convert to CI format
palette 0  =  2, RGB565             ; create palette 0 from image 2
                                    ; convert palette entries to RGB565 format
texture 2  =  2, 0                  ; texture 2[RGB] = image 2, texture 2[CLUT] = palette 0

path        = NULL                  ; no more path
file    3   =  c:/marioBody_i8.tga  ; file name now requires full path
image   3   =  3, x, I8, GX_REPEAT, GX_REPEAT ; TC will auto-convert from rgb input to intensity
                                               ; output.  Will set wrap mode in S and T to repeat.
                                               ; Can set wrap modes with mipmap arguments as well.
texture 3   =  3, x
```

Code 10 Texture converter script

2.2.2.1 General notes on script files

- White space is ignored; e.g., “image1=1” and “image 1 = 1” are the same thing.
- Text is parsed line by line. Lines end with a newline and must be < 255 characters.
- Images, textures, palettes, and their indices may be listed in any order. TC sorts each list into ascending order before conversion.
- Comments begin with a semicolon and continue to the end of a line.
- Missing components are indicated by an “x”; e.g., an image that lacks an alpha plane is described as “image 1 = 1, x, RGB565”, while a texture without a palette is described as “texture 1 = 1, x”.
- Directory pathnames may be absolute or relative to the current directory.

2.2.2.2 Notes on list order

In the script file: All lists are opened when script file parsing begins. Text lines are identified by their first word (e.g., “image,” “palette,” “texture”), and a new element is added to the appropriate list. Therefore, images, textures, palettes, and their indices may be listed in any order. In Code 10, images, palettes, and textures are listed in alternating order. It is also acceptable to list all images together in a single block, followed by all palettes, and so on. Once the script file is closed, TC sorts each list into ascending order.

In the TPL file: Images, textures, and palettes are packed into the TPL in the order in which they appear in their sorted lists. Normally, list indices will correspond to sorted list position (e.g., image 0 will be first in the list, and image (*n*-1) will be last). However, it may be convenient to comment out some elements of the script file when trying to determine an optimal TPL configuration. Once you have the final TPL configuration, indices should be renumbered so that they run continuously from 0 to (*n*-1). Although renumbering is not mandatory, we strongly recommend it in order to avoid confusion when trying to access textures at runtime.

path = *directory_path*

- *directory_path* is the directory containing the input files. *directory_path* is pre-concatenated to all subsequent file names. **path** remains in effect until changed. To turn **path** off, set **path** = 0 or **path** = `NULL`.
  path may be either absolute or relative to the current directory.

file *file_id = file_name*

- *file_id* is a number used by the **image** and **texture** commands to reference the file *file_name*. Valid numbers are 0 to any positive integer.
- If **path** is `NULL`, *file_name* must include a full pathname. The pathname may be absolute or relative to the current directory. If **path** is set, *directory_path* will be pre-concatenated to *file_name*.

image *image_id = rgb_image_file_id, alpha_image_file_id, format* [,*start_lod, end_lod, remap_lod*] [,*wrapS, wrapT*]

- *image_id* is a number used by the **texture** command to reference the image. Valid numbers are 0 to any positive integer.
- *rgb_image_file_id* is the file which contains the RGB portion of this image.
- *alpha_image_file_id* is the file which contains the alpha portion of this image.

As you can see, the **image** command is very powerful. Using *rgb_image_file_id* and *alpha_image_file_id*, you can compose an output RGBA image from two different source files. This feature is not often used with image formats that have full alpha plane capability (like TGA), but it is extremely useful with image formats that cannot store alpha information.

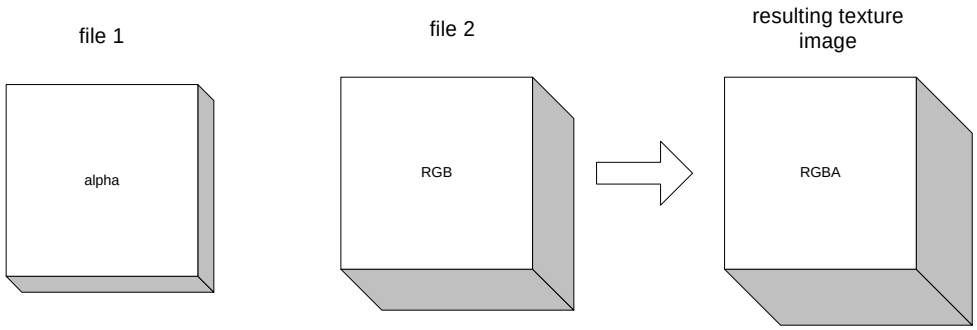

Figure 9 Creating an RGBA image from two different files

- *format* specifies the desired output GCN format. The possibilities include all non-Z texture formats in the GX library `GXTexFormats` enumerated type. We have removed the `GX_TF_` prefix in the script for simplicity.
- [,*start_lod, end_lod,remap_lod*] are optional arguments needed only for generating mipmaps. *start_lod* and *end_lod* specify which mipmap levels `TexConv` should generate. *remap_lod* specifies how to map to LOD at runtime. Typically, *remap_lod* will be 0.
- [,*wrapS, wrapT*] are additional optional arguments for overriding the default wrap modes for the *s* and *t* axes. The possible values are `GX_REPEAT`, `GX_CLAMP`, and `GX_MIRROR`. The default will be `GX_REPEAT` if both height and width dimensions are a power of 2, and `GX_CLAMP` if otherwise.

The following example shows how to generate two LOD levels and re-map to **lod0** at runtime.

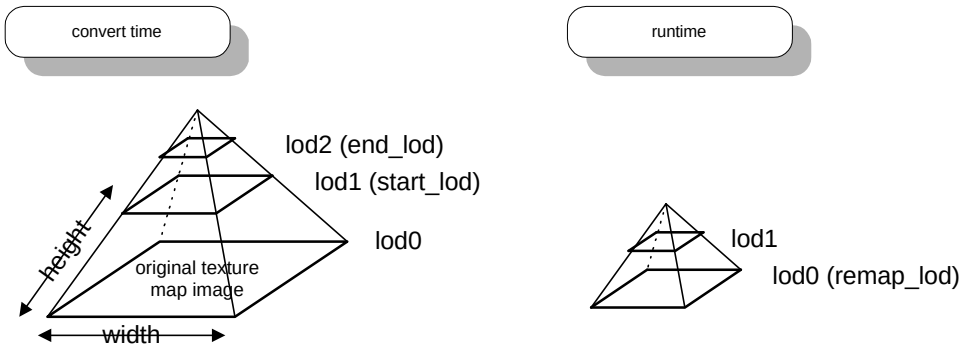

Figure 10 LOD level generation and re-mapping

palette *CLUT_palette_id = CLUT_file_id, dolphin_CLUT_format*

Supported palette entry formats are RGB565 and RGB5A3.

texture *texture_id = image_id, CLUT_palette_id*

The **texture** command actually defines a texture in the TPL file. The value of *texture_id* must be from 0 to any positive integer.

- *image_id* defines the image that represents this texture.
- *CLUT_palette_id* defines the CLUT palette used for this texture. For non-color-indexed textures, set this parameter to 'x'.

2.3 TC API

2.3.1 Overview

The TC library provides a public API to enable the following features:

- Limit the number of open files resident in memory.
- Install a user-defined file reader function.
- Create the components of TC's internal file structure (`TCFile`).
- Create a TPL file.

2.3.2 Source files

For the full structure definitions, function prototypes, and source code of the public API, see:

```
/cp/include/charPipeline/tc   (headers)
/cp/build/libraries/tc/src    (source)
```

2.3.3 APIs

To conserve memory, TC limits the number of source image files that may be resident in memory at any given time. A file in memory is stored as a "`TCFile`" structure. A `TCFile` is a simplified form of an image file, consisting of an RGB or indexed color plane, an optional alpha plane, and an optional palette. Caching a file in `TCFile` format saves the time required to fetch and decode the raw file on subsequent loads.

2.3.3.1 TCSetFileCacheSize

```
void TCSetFileCacheSize ( u32 size );
```

Code 11 TCSetFileCacheSize

`TCSetFileCacheSize()` allows the user to define the number of elements in this file cache. Maximum cache size varies depending on input file sizes, TC's internal memory allocation requirements, and available RAM.

Use of this function is optional. If it is not called, the cache defaults to size = 1.

2.3.3.2 TCInstallFileReadFn

We expect that artists will create source images in a variety of formats. Since we cannot support all possibilities, TC includes a plug-in mechanism to implement file reading functions. This allows the tool programmer to define and install file readers for any input file type desired.

```
void TCInstallFileReadFn( char* ext, u32 (*fileFn)( u32 rawSize, u8* rawBits, TCFile* fPtr ) );
```

Code 12 TCInstallFileReadFn

There is no limit to the number of file readers that may be installed. To install multiple readers, call `TCInstallFileReadFn` as many times as necessary with different "*ext*" and "*fileFn*" parameters.

For example, to install two file readers:

```
TCInstallFileReadFn( “TGA”, pTgaFn );
TCInstallFileReadFn( “BMP”, pBmpFn );
```

NOTE: `TexConv` includes a default TGA file reader that installs with a call to this function. The source code is available in:

```
dolphin\build\graphicTool\TexConv\src\tga.cpp
```

2.3.3.3 File reader APIs

TC registers file types in a table of 3-letter file extensions and function pointers. When TC reads a file, it first searches this table for a matching extension and calls the corresponding function. TC provides the raw file size, a pointer to the raw file bits and an empty `TCFile` structure. The file reader is responsible for unpacking the raw data into the `TCFile`.

The `TCFile` splits the source image into three components:

- Color layer (RGB or CI).
- Alpha layer.
- Palette.

The user-defined file reader is responsible for initializing and filling each relevant component.

```
TCLayer* TCCreateLayer    ( void );
void TCSetLayerAttributes ( TCLayer* ly, u32 type, u32 width, u32 height );
u8*  TCSetLayerBuffer     ( TCLayer* ly );
void TCSetLayerValue      ( TCLayer* ly, u32 x, u32 y, u16 ria, u8  g, u8 b );
void TCGetLayerValue      ( TCLayer* ly, u32 x, u32 y, u16* riaPtr, u8* gPtr, u8* bPtr );

TCPalTable* TCCreatePalTable ( u32 numEntry );
void TCSetPalTableValue   ( TCPalTable* ptPtr, u32 index, u8 r, u8 g, u8 b, u8 a );
void TCGetPalTableValue   ( TCPalTable* ptPtr, u32 index, u8* rPtr, u8* gPtr, u8* bPtr, u8* Ptr );
```

Code 13 File reader APIs

A layer (`TCLayer`) stores image data as 24-bit RGB, 8-bit alpha or 16-bit color-indexed (CI) texels. Texel data is stored in uncompressed format, row-major order, with the top left corner of the image as texel (0,0).

A palette (`TCPalTable`) contains an array of 32-bit RGBA color values. Palette array indices map directly to the `TCFile`’s corresponding CI color layer indices.

The file reader APIs shield the programmer from layer and palette internals. You create a layer by calling `TCCreateLayer`, `TCSetLayerAttributes` and `TCSetLayerBuffer`. After that, the data is filled in one texel at a time by (width *x* * height *y*) calls to `TCSetLayerValue`. Depending on its type, `TCSetLayerValue` interprets “*ria*” to mean “red, index or alpha.” If the *g* and *b* parameters are not required, they are ignored.

You can create and fill palettes (`TCPalTable`) in a similar manner; i.e., by calling `TCCreatePalTable` followed by *numEntry* calls to `TCSetPalTableValue`.

NOTE: If an input palette lacks alpha, all `TCFile` palette alphas must be set to 255.

For detailed information about writing a file reader, see section 4.5.

2.3.3.4 Creating a TPL file

You can create a TPL file with a single function call. Pass in the name of the TCS file and the name of the desired output file.

```
void TCCreateTplFile( char* srcTxtFile, char* dstTplFile );
```

Code 14 TCCreateTplFile

TC uses most of the public API to implement the user-defined file reading functions. Once you've created a file reader, using TC is very simple.

The following is a sample program to invoke TC. Assume that *pFileFn* is a user-defined file reader, and that *srcTcsFile* and *dstTplFile* are input and output file names.

```
void main()
{
    TCSetFileCacheSize( 1 );
    TCInstallFileReadFn( "TGA", pFileFn );
    TCCreateTplFile( srcTcsFile, dstTplFile );
}
```

Code 15 Sample TC main function

2.3.4 A note about error messages

Non-recoverable errors trigger a `printf`-style message and terminate the program. Depending on where the error occurred, the new TPL file and CSF file may be corrupt. After a fatal error, delete the CSF file (`C:\Temp\*.csf`) or TPL file to ensure full reconversion the next time.

Error messages appear in the console window in the form "`error: <function name> <message>`".

- `<message>` provides a short description of the error condition and should be sufficient to identify common problems.
- `<function name>` refers to the internal function where the error occurred. This can help programmers track bugs in cases where `<message>` does not provide sufficient information.

2.4 Output

2.4.1 TPL file

For a byte-by-byte description of the TPL format, please refer to section XXX.

2.4.2 Cache file and format

We expect that an artist will iterate many times through the "edit–convert–preview" loop when fine-tuning textures. In this situation, most of the textures in a TPL file will remain unchanged between successive conversions. TC saves conversion time by block-copying the unchanged textures from the previous TPL, then reconverting only those textures that have been altered since the last time the file was run.

To determine if an image or palette requires conversion, TC maintains a cached file (CSF) describing the contents of the previous TPL. TC uses the CSF file, previous TPL file, and source image files to identify and copy pre-existing data.

The CSF file is stored as `C:\Temp\tplCache.csf`.

In general, it should not be necessary to read or modify the CSF file. However, in case of need, the CSF file format is fully described by the comments and structures at the top of the following source file:

```
/cp/build/libraries/tc/src/TCTPLToolBox.cpp
```

The following situations will trigger full reconversion:

- Missing `.csf` file.
- Missing previous `.tpl` file.
- Previous `.tpl` has been modified since it was created.
- Previous `.tpl` version number does not match current "code" version number.
- Previous `.tpl` and new `.tpl` have no images or palettes in common.

When deciding whether to convert or copy data blocks, TC asks, "Does the previous `.tpl` contain an image/palette block with the same filename and attributes as the image/palette to be converted?" This means that script file indices and the `.tpl` filename may be changed between conversions while still leveraging pre-converted data.

The following situations will trigger partial reconversion:

Condition	TC Action
Change to source image file modification time.	Reconvert any image, palette referencing this file.
Change to script file contents.	Reconvert affected images, palettes.
Addition to script file contents.	Convert any new images, palettes.

Table 4 Partial reconversion parameters

NOTE: When deciding whether data can be re-used, TC checks script file tokens and file modification dates; it does not check actual source image bits. If any conversion code has been rewritten between successive calls to TC, this may cause errors because previous TPL data will be out of sync with current conversion code. To force a full reconversion, delete either the `.csf` file or the previous `.tpl` file.

2.5 Demonstration

Files for the texture converter demo are provided in the following directory:

```
/cp/build/tools/TexConv/sample
```

The demo shows how to write a script to generate a TPL file. To run this demo to generate a sample TPL file, type

```
cd /cp/build/tools/TexConv/sample
/cp/x86/bin/TexConv sample.tcs sample.tpl
```

After conversion, the result will be stored in `sample.tpl`.

To view the TPL files generated by `TexConv`, you can utilize the texture previewer. The **texture previewer** is called `texPrev2` and should not to be confused with the GCN previewer that can preview geometry files. The texture previewer is located in

```
/cp/build/demos/texPrev2/bin/MAC/texPrev2D.bin
```


Copy this file to a PowerMac G3 computer. Double-click on this file to launch StuffIt Expander to create the executable. When you execute the texture previewer, directions should appear in the text box below the viewport.

3 Extending the C3 library

We created the C3 library in order to demonstrate an example of a generic tool path optimized for the GCN hardware; however, the converter library's feature set is very basic, so providing a simple method of extension was crucial. The C3 library allows programmers to attach their own data in one batch to each of the GPL, ACT, and ANM formats:

```
void C3SetGPLUserData ( u32 size, void *data );
void C3SetACTUserData ( u32 size, void *data );
void C3SetANMUserData ( u32 size, void *data );
```

Code 16 C3 user data API

A corresponding API exists for retrieving the size and data pointer in the C3 library. The user data begins in its appropriate format, starting at a 32-**byte**-aligned boundary. It is padded to a 32-**bit** boundary at the end. Keep in mind that you may have to swap bytes to ensure the correct endian format, depending on your data structures.

4 Extending the texture converter libraries

You may find it necessary to modify the TC library to provide additional functionality. While this document cannot cover every scenario, sections 4.1 to 4.4 discuss how and where to integrate new code for the following most likely types of extensions:

- Changes to mipmap filter.
- Improved file caching.
- Palette generation.
- Mipmapping a color-indexed image.

Section 4.5 explains how to write and install a file-reading function for image formats other than TGA.

4.1 Changing mipmap filter, color computations

TC's current mipmapping algorithm is fully described in Appendix C. This algorithm assumes that alpha is used to indicate a texel's transparency value. RGB color components are treated as a single channel, and color is pre-multiplied by alpha.

To use color or alpha in some other manner:

- Edit the mipmapping functions `TCCreateNextMipMapLayer` and `TCBoxFilter` in `/cp/build/libraries/tc/src/TCMipMap.cpp`.
- Rebuild both TC and `TexConv`.

4.2 Improving file caching

TC's current file cache scheme is primitive. The file cache consists of an array of entries. Each time a file is opened, TC checks for a free entry. If it finds one, it stores the new file in the free element. Otherwise, the new file overwrites entry 0.

Because TPL files rarely re-use image files, it is difficult to devise a more sophisticated cache-management algorithm. One possibility is a "least-recently used" algorithm, but a better option may be to create a number of "locked" cache elements for frequently used files (i.e., they load once and stay resident) and designate a single entry as a buffer into which "one-time use" files are loaded in turn.

File loading and management code is located in

```
/cp/build/libraries/tc/src/TCFile.cpp
```

4.3 Generating palettes

Although TC does not provide any mechanism for generating palettes from true-color images, there are a number of well-known algorithms available to perform color reduction. However, integrating color reduction code directly into TC's conversion loop may be difficult. A better approach is to perform the operation from within the user-defined file-reading function at file load time. (For information on writing a file reader, see section 4.5.)

NOTE: `WriteTplImageBank` in `TCTPLToolbox.cpp` contains a switch statement preventing CI input to RGB output. Edit this statement to remove the error flag.

4.4 Mipmapping a color-indexed image

Currently, TC generates an error if an attempt is made to mipmap a CI image. To disable this error, edit the switch statement in `TCCheckMipMapFormats` located in

```
/cp/build/libraries/tc/src/TCMipMap.cpp
```

To create CI mipmaps, add the appropriate code to `TCCreateNextMipMapLayer` in the same file.

NOTE: TPL files support a maximum of only one palette per CI image, regardless of the number of LODs created. Therefore, all LODs of a CI image will be forced to use the base palette for their color selections.

4.5 Writing a file reader

`TexConv` supplies a file reader for TGA files. Studying the source code should help to clarify the issues covered in this section. You can find the file reader source code in

```
/cp/build/tools/TexConv/src/tga.cpp
```

The file reader is responsible for unpacking raw image bits into a `TCFile` structure. A `TCFile` is TC's internal representation of a file. It stores color and palette data in a simplified, modular, non-lossy format.

You must write a separate file reading function for each additional image file format you wish to support. The file reader function must have this prototype:

```
u32 fileFn( u32 rawSize, u8* rawBits, TCFile* fPtr );
```

TC provides the *rawSize*, *rawBits* and *fPtr* parameters.

- *rawSize* is the size of the raw file in bytes.
- *rawBits* is a buffer containing the raw file data.
- *fPtr* is a pointer to a zeroed-out element of TC's internal file cache.

fileFn must return 1 on success, 0 on failure.

The public portion of a `TCFile` structure looks like this:

```
typedef struct
{
        // … other members

        TCLayer*    lyColor;       // image color layer
        TCLayer*    lyAlpha;       // image alpha layer

        TCPalTable* palPtr;        // palette

} TCFile, *TCFilePtr;
```

Code 17 TCFile structure

A `TCFile` contains other data members, but they are for TC's internal use only. When *fPtr* is passed to *fileFn*, all of its public pointers are set to `NULL`. *fileFn* must set the *lyColor* member; *lyAlpha* and *palPtr* are optional. *fileFn* may safely ignore the pointer for any layer or palette it does not intend to set.

TC provides the input file as a buffer full of raw data, and an API to create and fill the components of *fPtr*. Therefore, *fileFn* does not require any file I/O or memory allocation operations.

Unpacking an image file is the process of separating its data into three components:

- Color layer.
- Alpha layer
- Palette.

4.5.1 TCLayer

A layer (`TCLayer`) is a set of like components from an image color map. A layer may contain RGB colors, alpha values, or color indices. A layer's type and dimensions are defined by `TCSetLayerAttributes`.

- The #defined type of a color layer is `LY_IMAGE_COLOR_RGB24`.
- A color-index layer is `LY_IMAGE_COLOR_CI16`.
- An alpha layer is `LY_IMAGE_ALPHA_A8`.

To represent an image in intensity/grayscale format, create an RGB color layer and set ($r = g = b$) for each texel. TC takes the sum of color components and divides it by three when converting to I or IA output formats.

A layer contains image data in uncompressed format. Image data begins with the top left corner of the image at texel (0,0) and proceeds in row order.

RGB colors are stored as 24 bits per texel; unsigned 8-bit values for each of *r*, *g* and *b*. Color indices are stored as unsigned 16-bit values per texel, with only the lower 14 bits used. Alpha is stored as an unsigned 8-bit value per texel.

Every source image must set a color layer. If the source image has alpha information, it must also set an alpha layer.

4.5.2 TCPalTable

`TCPalTable` is the palette from a source image. A `TCPalTable` contains an array of palette entries, and an entry's array index equals its colormap index. For example, if palette `entry[5]` is r=0, g=0, b=255, a=255, and the value of texel (m, n) is 5, then the color at texel (m, n) is blue.

- Entries are stored as 32-bit RGBA values; i.e., as unsigned 8-bit values for each for *r*, *g*, *b* and alpha.
- If the input palette has no alpha, each `TCPalTable` entry must set alpha to 255.

4.5.3 Unpacking an image file

It is not necessary to know `TCLayer` or `TCPalTable` internals to unpack a file because TC provides an API for complete layer/palette management.

To create and fill *fPtr*'s layers:

- Call `TCCreateLayer` to allocate a new `TCLayer`.
- Call `TCSetLayerAttributes` to set the new layer's width, height and type ("type" tells TC how many bits per texel this layer's data will require; for #defined type values, see section 4.5.1 above).
- Call `TCSetLayerBuffer` to allocate a data buffer.
- Set texel values individually with (width*height) calls to `TCSetLayerValue`. If a color component is not required (e.g., *g* and *b* for an alpha layer), pass 0.

To create and fill *fPtr*'s palette:

- A call to `TCCreatePalTable` will both allocate and initialize a new `TCPalTable`. Pass in the number of entries required.
- Set individual palette entries with (*numEntry*) calls to `TCSetPalTableValue`. If the input palette has no alpha, be sure to set each entry's alpha value to 255. Otherwise, if the output palette format is RGB5A3, the alpha component will contain garbage.

NOTE: *fileFn* does not know the intended output format of a source image. In cases where TC must auto-convert from a CI input image to an RGBA output format, only the color layer is first converted from color-indexed to an intermediate RGB format—the alpha layer is passed straight through. This is a by-product of TC's image-gluing scheme whereby output color and alpha can be derived from separate source images. Therefore, you have a color-indexed image and its palette has an alpha component, *you must preserve alpha both in an alpha layer and in the palette*. Create a true alpha layer for *fPtr->lyAlpha* by performing the color lookup for each texel, then set the input palette alpha in the new *fPtr->palPtr*.

Here is a simple code example:

```
/*
API structures, #defines, prototypes are located in dolphin/include/charPipeline/tc
Working code is located in dolphin/build/graphicTools/TexConv/src/tga.cpp

user-defined structures, functions, variables for example fileFn:

myFileInfo:            define a structure to store header information from 'rawBits'
myLayerType:           true-color (rgb) layer is LY_IMAGE_COLOR_RGB24
                       color-index layer is      LY_IMAGE_COLOR_CI16
                       alpha layer is            LY_IMAGE_ALPHA_A8.
myGetColorPlaneType(): determines 'myLayerType' value from 'rawBits'

myGetTexelColor():     fetches r,g,b values for texel ( x, y ) from 'rawBits'.
                       If 'rawBits' is intensity format, must set r = g = b.
myGetTexelAlpha():     fetches 8-bit alpha value for texel ( x, y ) in 'rawBits'.
myGetPaletteEntry():   fetches rgba value from 'rawBits' for CLUT entry.
                       If input palette has no alpha, must set 'a' to 0xFF.
*/

// sample file reader
u32 fileFn( u32 rawSize, u8* rawBits, TCFile* fPtr )
{
     struct myFileInfo;       // user-defined header for 'rawBits'
     u32 myLayerType; // defines layer type – rgb color, color-index or alpha
     u16 ria;                 // "red, index or alpha":
                             // = 8b red component of an rgb texel       (color layer)
                             // or 16b index of a color-index texel      (color layer)
                             // or  8b value of an alpha texel           (alpha layer)
     u8 g, b;                // green, blue components of an rgb texel   (color layer)


    // each source file must have a color layer
    fPtr->lyColor = TCCreateLayer();

    // determine 'rawBits' color layer type.
    myLayerType = myGetColorPlaneType( &myFileInfo ); // one of LY_IMAGE_COLOR_RGB24,
                                                      //        LY_IMAGE_COLOR_CI16
    // initialize TCLayer members
    TCSetLayerAttributes( fPtr->lyColor,     myLayerType,
                          myFileInfo->width, myFileInfo->height );

    // allocate layer data buffer
    TCSetLayerBuffer( fPtr->lyColor );

    // translate one texel at a time:
    // read from 'rawBits'; write to fPtr->lyColor
    for( row = 0; row < myFileInfo->height; row++ )
```

```
    {
        for( col = 0; col < myFileInfo->width; col++ )
        {
            // note: if rawBits is intensity format, myGetTexelColor must set (r = g = b)

            myGetTexelColor( rawBits,       col, row, &r, &g, &b );
            TCSetLayerValue( fPtr->lyColor, col, row,  r,  g,  b );
        }
    } // done

    // alpha layer is optional: if no alpha in rawBits, skip this step.
    // note:  if rawBits contains a palette with alpha, must create a true alpha
    // layer for this file.
    if( myFileInfo->hasAlpha )
    {
        // create and fill as for color layer,
        // but set myLayerType = LY_IMAGE_ALPHA_A8,
        // write myGetTexelAlpha to fetch 'a' from rawBits
        // and call TCSetLayerValue with ( …, a, 0, 0 )
    }

    // palette is optional: if no palette, skip this step
    if( myFileInfo->hasPalette )
    {
        // call fPtr->palPtr = TCCreatePalette( myFileInfo->palNumEntry );
        // loop (for entry = 0; entry < myFileInfo->palNumEntry; entry++)
        // call TCSetPalTableValue( …, entry, … ) to set palette entries.

        // note: if input palette lacks alpha, must set 'a' to 0xFF for each entry.
    }

// note: do NOT free rawBits

    return 1;  // success
}
```

Code 18 Sample file reader

Appendix A. Building source code

A.1 Building `CPExport.dle` for 3D Studio MAX Release 3.1

The 3D Studio MAX Release 3.1 plug-in converter requires Microsoft's Visual C++ 6.0 compiler with Service Pack 3. Before you can build `CPExport.dle`, please complete the following steps:

1. Make sure that the NINTENDO GAMECUBE (GCN) SDK is installed (required to access additional CP libraries that could not be separated into the CP SDK).
2. Install the SDK for 3D Studio MAX Release 3.1, which is located on the 3D Studio MAX Release 3.1 Installation CD.
3. Copy the files `Bipexp.h` and `Phyexp.h` from the `[MAX R3 Install CD]/Cstudio/Cstudio/Docs` directory to the `[Maxsdk]/Include` directory.
4. Launch Microsoft Visual C++ 6.0 and open this workspace: `/cp/build/tools/Max/CPExport/vc++/MaxConv.dsw`.
5. If the 3D Studio MAX SDK is installed in the default path "`C:\3dsmax3_1\Maxsdk`", skip to step 10.
6. In Visual C++, change to "File View" in the Workspace Window.
7. Right-click on "MaxConv files" and select "Settings…". Make sure the "Settings For:" drop-down list shows "All Configurations".
8. Click on the "C/C++" tab. In the "Category" drop-down list, select "Preprocessor". Under "Additional include directories", add the full path to your `[Maxsdk]/include` directory (installed when you completed step 1). This directory may be hiding at the end of the line.
9. Click on the "Link" tab. In the "Category" drop-down list, select "Input". Under "Additional library path", add the full path to your `[Maxsdk]/lib` directory. Close the dialog.
10. Finally, click on Build > Build CPExport.dle. `CPExport.dle` will be located in `/cp/x86/lib`.

To learn how to install `CPExport.dle` as an export plug-in to 3D Studio MAX, please refer to Chapter 1 of "Character Pipeline for Artists." The debug version of the export plug-in will be named `CPExportD.dle`.

You can build both release and debug versions for the 3D Studio MAX converter. Make sure to set the active project configuration for "Win32 Release" or "Win32 Debug," respectively, for all projects under Build > Set Active Configuration…

NOTE: The CP SDK installer makes sure that programs are in their correct directories. For your information, `CPExport.dle` will call `TexConv.exe` to generate the TPL file for the exported scene. It expects `TexConv.exe` to be in the directory specified by the environment variable `CP_X86_BIN`, which should be installed by the CP SDK.

A.2 Building `CPExport.mll` for Maya 3.0

The Maya 3.0 export plug-in also requires Microsoft's Visual C++ 6.0 with Service Pack 3. Please follow the following steps to set up the build process:

1. Make sure that the NINTENDO GAMECUBE (GCN) SDK is installed (required to access additional CP libraries that could not be separated into the CP SDK).
2. Install Maya 3.0.

3. Launch Microsoft Visual C++ 6.0 and open this workspace: `/cp/build/tools/Maya/CPExport/vc++/MayaConv.dsw`.
4. If Maya is installed in the default path "`C:\AW\Maya3.0`", skip to step 9.
5. In Visual C++, change to "File View" in the Workspace Window.
6. Right-click on "MayaConv files" and select "Settings…". Make sure the "Settings For:" drop-down list shows "All Configurations".
7. Click on the "C/C++" tab. In the "Category" drop-down list, select "Preprocessor". Under "Additional include directories", add the full path to your `Maya3.0/include` folder and `Maya3.0/devkit/games/include` folder. These directories may be hiding at the end of the line.
8. Click on the "Link" tab. In the "Category" drop-down list, select "Input". Under "Additional library path:", add the directory to `Maya3.0/lib`. Close the dialog.
9. Finally, click on Build > Build CPExport.mll. `CPExport.mll` will be located in `/cp/x86/lib`.

To learn how to install `CPExport.mll` as an export plug-in to 3D Studio MAX, refer to Chapter 1 of "Character Pipeline for Artists."

You can build both release and debug versions for the Maya converter. Make sure to set the active project configuration for "Win32 Release" or "Win32 Debug,"respectively, for all projects under Build > Set Active Configuration… The debug version of the export plug-in will be named `CPExportD.mll`.

NOTE: The CP SDK installer makes sure that programs are in their correct directories. For your information, `CPExport.mll` will call `TexConv.exe` to generate the TPL file for the exported scene. It expects `TexConv.exe` to be in the directory specified by the environment variable `CP_X86_BIN`, which should be installed by the CP SDK.

A.3 Building `TexConv.exe`

1. In Visual C++, open `/cp/build/tools/TexConv/vc++/TexConv.dsw`.
2. Click on Build > Rebuild All. The executable will be located in the `/cp/x86/bin` directory.

For a demonstration of the `TexConv` converter on its own, refer to section 2.5.

Appendix B. Texture compression

B.1 S3 library

TC uses `s3tc.lib` to generate compressed (CMPR) textures. Texture compression is a two-step process:

1. `s3tc.lib` generates compressed textures in its own `S3_TEXTURE` format.
2. The compressed textures are modified to fit the hardware format (see section B.2 below) and copied to the new TPL file.

Source code for this conversion is located in

```
/cp/build/libraries/tc/src/TCCreateS3.cpp
```

Compressed textures are generated one LOD at a time. Note that `s3tc.lib` does not generate mipmaps; the calling code must first generate LODs, which are then passed one at a time to the S3 conversion API.

B.1.1 Complete S3 library documentation

`s3tc.lib` is located in

```
/cp/x86/lib
```

The library's header file (`s3_intrf.h`) is located in

```
/cp/build/libraries/tc/include
```

S3's own documentation and copies of the S3 libraries for PC, Mac and Linux are stored as a zip file in

```
/cp/build/libraries/tc/s3tclib/S3TCLib.zip
```

Printed S3 documentation is included in the *NINTENDO GAMECUBE Graphics Programmer's Guide*.

B.2 Converting an S3 texture to hardware format

When packing an S3 texture for conversion to hardware format, there are four major points to consider:

- The S3 library converts a source texture into 4x4 texel tiles. Each tile contains two 16-bit colors and sixteen 2-bit texel indices for a total of eight bytes. The tiles are packed in row-major order. The hardware requires that these tiles be packed as 2x2 "super" tiles. Each super tile consists of four S3 tiles for a total of 8x8 texels and 32 bytes:

Hardware Tile	S3 Tile Index	Image Texels
bytes (0-7)	= S3 Tile[0][0]	= texels (00, 01, 02, 03), (10, 11, 12, 13), (20, 21, 22, 23), (30, 31, 32, 33)
bytes (8-15)	= S3 Tile[0][1]	= texels (04, 05, 06, 07), (14, 15, 16, 17), (24, 25, 26, 27), (34, 35, 36, 37)
bytes (16-23)	= S3 Tile[1][0]	= texels (40, 41, 42, 43), (50, 51, 52, 53), (60, 61, 62, 63), (70, 71, 72, 73)
bytes (24-31)	= S3 Tile[1][1]	= texels (44, 45, 46, 47), (54, 55, 56, 57), (64, 65, 66, 67), (74, 75, 76, 77)

Table 5 Super tile composition

- Bytes (0-3) of each 4x4 S3 tile contain two 2-byte RGB colors. These colors are packed in little-endian format, but for hardware, these colors must be packed in big-endian format. Therefore, each 2-byte color must have its byte order switched.

- Bytes (4-7) of each 4x4 S3 tile contain 4 sets of row indices. Each set occupies one byte. Each set contains four 2-bit indices, one for each texel in the row. The ordering of these indices must be reversed within each byte; however, bit order remains the same within each 2-bit index. For example, an input byte of 0x1B becomes 0xE4 on output.

- The dimensions of an S3 texture may be any multiple of four texels. Hardware requires that super tiles be formed in multiples of 8x8 texels. That is, if any of the S3 texture's dimensions is not a multiple of 8 texels, it will be necessary to pad the output image with zeroed-out tiles at the right and/or bottom edges.

B.2.1 Further reading

For diagrams of texel formats, see "Graphics Library (GX)," Appendix C, in the *NINTENDO GAMECUBE Graphics Programmer's Guide*.

For the full `TC.lib` source code, please refer to the functions `TCWriteTplImage_CMP`, `TCPackTile_CMP`, `TCFixEndian` and `TCFixCMPWord` located in

```
/cp/build/libraries/tc/src/TCTPLToolbox.cpp
```

B.2.2 Code example

The following code sample demonstrates how to pack an S3 texture. For clarity, there is no checking of S3 image dimensions for 8-texel padding, and some local variable definitions have been omitted.

```
// pack an S3-compressed texture into a set of 32-byte hw tiles
void PackS3Texture( u8* s3Texture, u32 s3Width, u32 s3Height, u8* outBuffer )
{
    u8* thisTile2x2;                    // ptr to top,left of 2x2 S3 tiles (8x8 texels)
    u32 s3StrideX = 16;                 // 2 S3 tiles are 8 texels wide at 8B per 4x4 tile
    u32 s3StrideY = s3Width / 4 * 8;    // 1 S3 tile is 4 texels high at 8B per 4x4 tile

    for( row = 0; row < ( s3Height / 4 ); row ++ )
    {
        for( col = 0; col < (s3Width / 4 ); col ++ )
        {
           thisTile2x2 = s3Texture + ( row * s3StrideY * 2 ) + ( col * s3StrideX );

           Pack32ByteTile( thisTile2x2 , s3strideY, outBuffer );

           outBuffer += 32;
       }
    }
}

// pack 2x2 S3 tiles into a single 32-byte hw tile
void Pack32ByteTile( p2x2Tile , s3strideY, outPtr )
{
    u8  tmpBlock[32];
    u8* thisTile;

    Copy8Bytes( ( p2x2Tile )               , ( tmpBlock )      );  // S3 tile at [0][0]
    Copy8Bytes( ( p2x2Tile + 8 )           , ( tmpBlock + 8 )  );  // S3 tile at [0][1]
    Copy8Bytes( ( p2x2Tile + s3StrideY )   , ( tmpBlock + 16 ) );  // S3 tile at [1][0]
    Copy8Bytes( ( p2x2Tile + s3StrideY + 8 ), ( tmpBlock + 24 ) );  // S3 tile at [1][1]
```



```
    thisTile = tmpBlock;
    for( i = 0; i < 4; i++ )                        // adjust bytes and bits within each S3 tile
    {
        SwitchWordEndian( (u16*)thisTile         );  // 1st 2-byte color
        SwitchWordEndian( (u16*)( thisTile + 2 ) );  // 2nd 2-byte color

        SwitchByteTuples( ( thisTile + 4 )  );       // 1st row of 4 texels
        SwitchByteTuples( ( thisTile + 5 )  );       // 2nd row of 4 texels
        SwitchByteTuples( ( thisTile + 6 )  );       // 3rd row of 4 texels
        SwitchByteTuples( ( thisTile + 7 )  );       // 4th row of 4 texels

        thisTile += 8;                               // advance 8 bytes within 32-byte block
    }

    Copy32Bytes( tmpBlock, outPtr );                 // copy the modified 32-byte block
                                                     // to the output file
}

// reverse 2-bit tuple ordering within a byte
void SwitchTuples( u8* pByte )
{

    *pByte = ( ( *pByte & 0x03 )  <<  6  ) | ( ( *pByte & 0x0C )  <<  2  ) |
             ( ( *pByte & 0x30 )  >>  2  ) | ( ( *pByte & 0xC0 )  >>  6  ) ;
}

// reverse endianness of a 2-byte word
void SwitchWord( u16* pWord )
{
    *pWord = ( ( *pWord & 0x00FF )  <<  8 ) | ( ( *pWord & 0xFF00 )  >>  8  );
}
```

Code 19 Packing an S3 texture for conversion

Appendix C. Mipmapping technical notes

There are several ways to construct mipmap LODs from an original (LOD 0) texture. Two important choices for such an algorithm are:

- The type of filter kernel used.
- The treatment of alpha values vs. color values.

The type of filter kernel used to construct less-detailed LODs from more-detailed ones has an effect on the final appearance of the textured object. A 2x2 filter kernel (i.e., equal-weight box filter) is common; indeed, we use it in our implementation. However, larger filter kernels (e.g., 4x4, 6x6, etc.) can be used to provide slightly better results. Larger filter kernels are not supported in the current implementation of mipmap LOD generation, but users can generate their own mipmap LODs if this feature is desired.

The treatment of alpha values vs. color values applies to textures that incorporate an alpha channel. The normal assumption for such a texture is that the alpha channel represents a transparency value to be applied to a given texel's RGB values. In order to generate less-detailed LODs correctly under this scenario, therefore, the alpha value for a given texel must be multiplied by the RGB values **before** averaging the various texels together, and the composite alpha value must be divided out again **before** writing the new RGB values. In essence, you compute a weighted-average for the new RGBs, with the alpha value serving as the weighting factor. Here's a visual example to explain.

Imagine that the following four texels must be combined to produce a single texel for a less-detailed LOD:

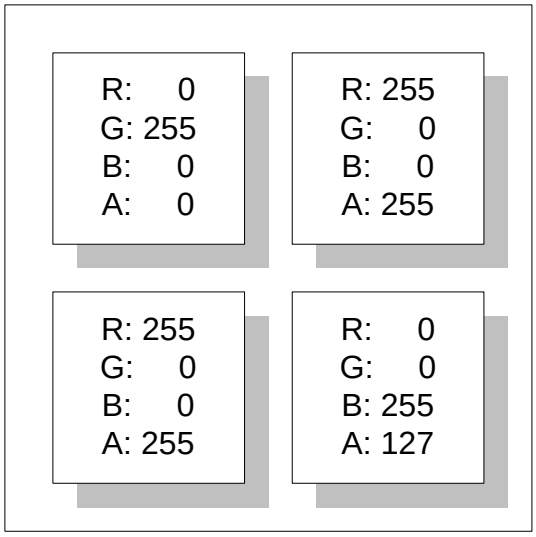

Figure 11 RGBA for LOD generation

- An alpha of 0 means the texel is transparent.
- An alpha of 255 means the texel is fully opaque.

The final texel should be computed like this:

$$A' = (0 + 255 + 255 + 127) / 4 = 159$$
$$R' = ((0 * 0 + 255 * 255 + 255 * 255 + 0 * 127) / 4) / A' = 204$$
$$G' = ((255 * 0 + 0 * 255 + 0 * 255 + 0 * 127) / 4) / A' = 0$$
$$B' = ((0 * 0 + 0 * 255 + 0 * 255 + 255 * 127) / 4) / A' = 51$$

Equation 1 Texture conversion (the right way)

And **not** like this (i.e., with RGBA treated separately):

$$R' = (0 + 255 + 255 + 0) / 4 = 128$$
$$G' = (255 + 0 + 0 + 0) / 4 = 64$$
$$B' = (0 + 0 + 0 + 255) / 4 = 64$$
$$A' = (0 + 255 + 255 + 127) / 4 = 159$$

Equation 2 Texture conversion (the wrong way)

Note that the upper-left original texel is the only one with any green in it. However, it is also completely transparent, and therefore the new texel should have no green in it (as the correct result shows). Similar logic applies to the other color components.

All of this is based upon the assumption that the alpha value represents the texel transparency. This assumption is made by the current LOD-generation implementation, and thus we use computations such as those shown in Equation 1. However, it is worth noting that there may be cases in which the alpha value for a given texture does not represent transparency. If alpha is to be used in some other manner, then the computations shown in Equation 1 may not apply. In such cases, the user should generate his own mipmap LODs.

For Equation 1, a special case for the box filter exists when `A'` is zero (i.e., all alpha values for the four texels are zero). Following the equation, if `A'` is zero, then `R'`, `G'`, and `B'` are undefined because you cannot divide by zero. Therefore, `R'`, `G'`, and `B'` were previously set to zero (black) as well. Now, in this special case, `R'`, `G'`, and `B'` will simply be an average of the `R`, `G`, and `B` values for the four texels, respectively. Since `A'` is still zero, this new texel is still considered to be transparent, but the color values are preserved.

Appendix D. AnmCombine

`AnmCombine` is a utility application that combines multiple ANM files into one ANM file. You can load source and header files in Microsoft Visual C++ 6.0 from:

```
/cp/build/tools/AnmCombine/vc++/AnmCombine.dsw
```

Building the application is simple. In Visual C++ 6.0, click on Build > Build AnmCombine.exe, or just press F7. `AnmCombine.exe` should be located in `/cp/x86/bin`.

`AnmCombine` is accessible through a command line interface:

```
AnmCombine.exe [ANM to combine to] [ANM file] [ANM file] ...
```

You can also specify a separate output file:

```
AnmCombine.exe [ANM file] [ANM file] ... /o [Output name]
```

All input ANM files must have the same byte endian, and the output byte endian is determined by the endian of the input files. We've provided an example in the `/cp/cpdata/max/Monkey` directory; please refer to the `README.txt` file for exact directions. `AnmCombine` will also ignore the user-defined data field in the ANM formats since the program does not know how it should combine them.

Appendix E. C3 library internal architecture

E.1 Introduction

This appendix further explains the internal data structures, algorithms, and program flow of the C3 library. We assume that you already understand the C3 API as explained in Chapter 1.

The C3 library was designed to separate the conversion paths for geometry, hierarchy, animation, and texture, thus we have four distinct output formats: GPL, ACT, ANM, and the TCS script. In keeping with the library's design, this document is divided into the following parts:

- Main program flow in section E.2.
- Geometry pipeline in section E.3.
- Hierarchy pipeline in section E.4.
- Animation pipeline in section E.5.
- Texture pipeline in section E.6.

Although we provide many code samples here, we recommend following along in the source code to familiarize yourself with the location of these major functional groups in the C3 library.

The C3 library is written in the C programming language and compiles using Microsoft Visual C++ 6.0 with Service Pack 3. A workspace file is provided in `/cp/build/libraries/c3/vc++/c3.dsw`, which provides access to all of the code:

- Public header files: `/cp/include/charPipeline/C3.h` and `/cp/include/charPipeline/c3/*.h`.
- Private header files: `/cp/build/libraries/c3/include/*.h`.
- Source files: `/cp/build/libraries/c3/src/*.c`.

Each source and header file carries a "C3" prefix and is named according to its purpose. For example:

- "`C3GeoCnv.c`" contains functions used to convert geometry.
- "`C3HieExt.c`" contains functions used to extract hierarchy information.
- "`C3AnmOut.c`" contains functions to output animation to the ANM format

Comments in the source code may also prove helpful in comprehending the C3 library.

E.2 Main program flow

The main program flow of the C3 library is straightforward. Section 1.1.3 in the main text explains the overall program flow using a code segment from the 3D Studio MAX converter. The following diagram shows each of the functional groups that we'll discuss here:

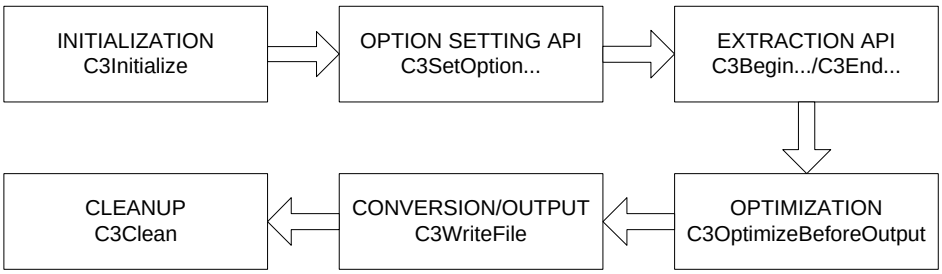

Figure 12 C3 library main program flow

E.2.1 C3Initialize

`C3Initialize` must be called first in order to initialize data structures and some important utilities. This function is located in `C3Util.c`.

E.2.1.1 Memory allocation

Memory allocation and deallocation in the C3 library may be done in one of two ways:

- `C3_CALLOC` and `C3_FREE`.
- Memory pools.

The first method is the traditional C-style method. `C3_CALLOC` and `C3_FREE` are macros that check memory usage and potential leaks in the C3 library, depending on whether `C3_CHECK_MEM_ALLOC` is enabled. While `C3_CHECK_MEM_ALLOC` is not defined by default in `C3Defines.h`, enabling it during compilation of the C3 library will track all memory allocation and deallocation. The results are provided in "`MemAlloc.txt`" in the export destination directory, with the file name and line number of allocation and deallocation. (See `C3Debug.h` and `C3Debug.c`.)

The second method uses memory pools. By pre-allocating a group of the same data structures, you can minimize memory fragmentation when many small allocations are necessary. However, there is a drawback in that no deallocation or reuse of this memory can occur until all the pools are cleared with `C3Clean` at the end. (See `C3Pool.h` and `C3Pool.c`.)

E.2.1.2 Structures library

Linked lists, trees, and hash tables are basic data structures used extensively in the C3 library, as well as in the rest of the Character Pipeline. Some of these types of structures are initialized during `C3Initialize`, but since linked lists are so important, their implementation in C3 should be explained before looking at any code.

The structures library is included with the NINTENDO GAMECUBE SDK, and not the CP SDK because the GCN SDK needs it for the texPalette and fileCache libraries. The structures library is located in

`/dolphin/build/charPipeline/structures`.

You can load the workspace in Microsoft Visual C++ 6.0 by opening

`/dolphin/build/charPipeline/structures/vc++/structures.dsw`.

The general-purpose linked list is located in `List.h` and `List.c`. Here is the structure declaration:

```
typedef struct
```



```
{
        Ptr     Prev;
        Ptr     Next;

} DSLink, *DSLinkPtr;

typedef struct
{
        u32     Offset;
        Ptr     Head;
        Ptr     Tail;

} DSList, *DSListPtr;
```

Code 20 DSLink and DSList

As you can see, `DSList` is the structure that contains a general pointer to the head (beginning) and tail (end) of its members. Also, all of the members in the list should be of the same data structure, and should include `DSLink`, which will doubly link the (next and previous) members in the list. Since `DSList` is general purpose, the `Offset` field in `DSList` is necessary when members are inserted and removed so that the `DSLink` pointers of the members are modified appropriately. Specifically, `Offset` is the number of bytes from the top of the member structure to the `DSLink` structure.

For example, let's create an example list and two members:

```
// Declare a new data structure to be member of a list
typedef struct
{
        // Your fields here (i.e. u32, Ptr, char*, etc.)
        // ...

        DSLink link;

        // More fields since DSLink can be anywhere in C3ListMember
        // ...

} C3ListMember;

// Data structures allocation
DSList         list;
C3ListMember   member1;
C3ListMember   member2;
C3ListMember  *memberPtr;

// Code sample

// Initialize the list for C3ListMember type (last two arguments used to initialize Offset)
DSInitList( &list, (Ptr)&member1, &member1.link );

// Insert member1 as first member, at top of list
DSInsertListObject( &list, NULL, (Ptr)&member1 );

// Insert member2 as second member, after member1
DSInsertListObject( &list, (Ptr)&member1, (Ptr)&member2 );

// Get the head of the list (should be member1)
memberPtr = (C3ListMember*)list.Head;

// Get the next member in the list using function (should be member2)
memberPtr = DSNextListObj( &list, (Ptr)memberPtr );

// Get the next member in the list manually (should be NULL)
memberPtr = (C3ListMember*)memberPtr->link.Next;
```

Code 21 Linked list example

As a result of Code 21, we should have a list that contains two members, as Figure 13 shows:

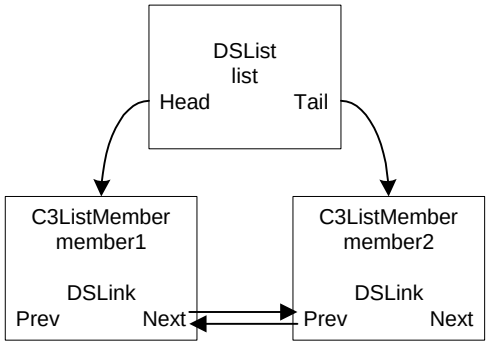

Figure 13 Schematic of linked list example (Code 21)

One advantage to this linked list structure is that it does not move memory; instead, it simply manipulates pointers. Note that although most applications require only the single link `Next`, the library offers only a doubly-linked list functionality for now.

The `DSTree` and `DSBranch` structures for specifying hierarchies rather than a flat list operate in a very similar manner. (See `Tree.h` and `Tree.c`.)

The `DSHashTable` structure implements a hash table using a list of `DSLists`. (See `HTable.h` and `HTable.c`.)

E.2.2 C3SetOption

`C3Initialize` sets default options, but you can set your own options in the C3 library by using the `C3SetOption*` functions in `C3Options.h` and `C3Options.c`. There is a brief overview in section 1.2.2, but here we discuss the purpose and show how to use each of the option-setting functions:

```
void C3SetOptionReportStatusFunc( C3ReportStatusFunc func )
```

- **Purpose:** Method by which the C3 library reports its current status.
- **Default:** `printf`.
- **Argument:** Function pointer with `void` return type and one argument with a `char*` type, as defined in `C3Util.h`. A `NULL` function will prevent any status messages.

```
void C3SetOptionReportErrorFunc( C3ReportErrorFunc func )
```

- **Purpose:** Method by which the C3 library reports any errors.
- **Default:** `printf`.
- **Argument:** Function pointer with `void` return type and one argument with a `char*` type, as defined in `C3Util.h`. A `NULL` function will prevent any error messages.

```
void C3SetOptionFileExportFlag( u32 fileExportflag )
```

- **Purpose:** Determines which files should be output: GPL, ACT, ANM, TCS script, or statistics file.
- **Default:** `C3_FILE_ALL`.
- **Argument:** Bitwise-OR combination of any of the following `C3Out.h` constants: `C3_FILE_GEOMETRY`, `C3_FILE_HIERARCHY`, `C3_FILE_ANIMATION`, `C3_FILE_TEXTURE`, or `C3_FILE_STATS`. For ease, `C3_FILE_ALL` can be specified to output all files.

`void C3SetOptionOutputEndian( u8 endianType )`

- **Purpose:** Sets the output endian, depending on destination hardware byte order.
- **Default:** `C3_OUTPUT_BIG_ENDIAN`.
- **Argument:** Either `C3_OUTPUT_BIG_ENDIAN` (for PowerPC/Mac and GCN) or `C3_OUTPUT_LITTLE_ENDIAN` (for PC).

`void C3SetOptionSrcVertexOrder( u8 vtxOrder )`

- **Purpose:** When extracting vertices, sets whether a front-facing polygon has a clockwise or counter-clockwise vertex order.
- **Default:** `C3_CCW`.
- **Argument:** Either `C3_CW` (clockwise) or `C3_CCW` (counter-clockwise).

`void C3SetOptionEnableStitching( C3Bool flag )`

- **Purpose:** Sets whether stitching or skinning is enabled in the C3 library. If not, all polygons will be rigid.
- **Default:** `C3_TRUE`.
- **Argument:** `C3_TRUE` or `C3_FALSE`.

`void C3SetOptionEnableLighting( C3Bool flag )`

- **Purpose:** Sets whether normals should be exported or not.
- **Default:** `C3_TRUE`.
- **Argument:** `C3_TRUE` or `C3_FALSE`.

`void C3SetOptionAmbientPercentage( f32 percent )`

- **Purpose:** Sets a percentage of the vertex color channel to be the ambient color in pre-lighting.
- **Default:** 25.0 percent.
- **Argument:** A percent between 0.0 and 100.0.

`void C3SetOptionCompress( u16 targets )`

- **Purpose:** Sets which vertex attribute arrays (position, normal, texture coordinates, vertex color) to compress, or removes duplicates.
- **Default:** All: `C3_TARGET_POSITION | C3_TARGET_NORMAL | C3_TARGET_TEXCOORD | C3_TARGET_NORMAL`.
- **Argument:** Bitwise-OR combination of any of the following `C3CnvOpz.h` constants: `C3_TARGET_POSITION | C3_TARGET_NORMAL | C3_TARGET_TEXCOORD | C3_TARGET_NORMAL`. **NOTE:** C3_TARGET_ALL does not include vertex color.

`void C3SetOptionWeldRadius( u32 target, f32 radius )`

- **Purpose:** Sets the welding threshold to combine vertex positions or texture coordinates within a given radius.
- **Default:** 0.0 for position and texture coordinates.
- **Argument:** *target* specifies vertex positions or texture coordinates. *radius* specifies welding radius.

`void C3SetOptionEnableStripFan( C3Bool flag )`

- **Purpose:** Sets whether triangle or quad polygons should be converted to triangle strips and fans.
- **Default:** `C3_TRUE`.
- **Argument:** `C3_TRUE` or `C3_FALSE`.

`void C3SetOptionEnableStripFanView( C3Bool flag )`

- **Purpose:** Sets whether polygon primitives should be wireframed and vertex colored depending on the type of the primitive for feedback. Red for triangles, Blue for quads, and random colors for strips and fans.

- **Default:** C3_FALSE.
- **Argument:** C3_TRUE or C3_FALSE.

```
void C3SetOptionPositionRange( f32 range )
```

- **Purpose:** Sets the position range for vertex position quantization other than exporting to float values. Given the range, the quantization scale will be calculated accordingly. Should be called before C3SetOptionQuantization for positions.
- **Default:** 0.0.
- **Argument:** *range* should be the absolute value of the largest *x*-, *y*-, or *z*-coordinate. Setting *range* to 0.0 will force the C3 library to compute the optimal position range, or quantization scale, for maximum precision.

```
void C3SetOptionQuantization( u32 target, u8 channel, u8 quantInfo )
```

- **Purpose:** Sets the quantization information for position, texture coordinates, normals and color (with and without alpha).
- **Default:** Positions, normals, and texture coordinates are set to GX_S16, while vertex color is quantized to GX_RGB565 and GX_RGBA4, if the object contains vertex alpha. Keyframes are not quantized (e.g., GX_F32).
- **Argument:** quantInfo specifies the quantization type and shift, and should always be set using the macro C3_MAKE_QUANT defined in C3Util.h. Only the type needs to be set, since the shift (or scale) is calculated automatically by the C3 library (see C3InitOptions in C3Options.c as an example). *channel* should always be 0, since this is used for future multitexture support. *target* specifies the vertex attribute according to the parameters below:
  - For C3_TARGET_POSITION, the range must be set earlier and properly by C3SetOptionPositionRange in order to calculate quantization shift bits.
  - For C3_TARGET_NORMAL, quantization shift bits are calculated assuming that all normals are normalized to length 1. If *channel* is not 0, this automatic calculation is turned off.
  - For C3_TARGET_TEXCOORD, the type of quantization is global, although the quantization scale for each object is computed separately for maximum precision.
  - For C3_TARGET_KEYFRAME, *quantInfo* should hold only the global type of quantization for all tracks. Only quantizes translation and scale and their IN and OUT controls. Quaternions are always quantized to GX_S16 since each *x*, *y*, *z*, and *w* component is between 0.0 and 1.0.
  - For C3_TARGET_COLOR, the type should be GX_RGB565, GX_RGB8, or GX_RGBX8. This quantization is used if the geometry object contains no vertex alpha.
  - For C3_TARGET_COLORALPHA, the type should be GX_RGBA4, GX_RGBA6, or GX_RGBA8. This quantization is used if the geometry object contains at least one vertex with a non-opaque alpha.

```
void C3SetOptionUseDefaultNormalTable( C3Bool flag )
```

- **Purpose:** Sets whether to use a default normal table of 252 normals instead of supplying a normal array for each geometry object.
- **Default:** C3_FALSE.
- **Argument:** C3_TRUE or C3_FALSE.

```
void C3SetOptionUseExternalNormalTable( C3Bool flag )
```

- **Purpose:** Sets whether to use a user-supplied external normal table in a similar manner to the default normal table. This function should be used in conjunction with C3SetOptionExternalNormalTablePath.
- **Default:** C3_FALSE.
- **Argument:** C3_TRUE or C3_FALSE.

```
void C3SetOptionExternalNormalTablePath( char* name )
```

- **Purpose:** If C3SetOptionUseExternalNormalTable is set to C3_TRUE, then a full path to the normal table should be specified.

- **Default:** `NULL`.
- **Argument:** *name* should be the full path to the external normal table; it cannot exceed `C3_MAX_PATH` in filename length.

E.2.3 C3Begin/C3End

After you've set all the desired options, you can extract data from a CG tool into the C3 library using the extraction API. Detailed explanations of the hierarchical C3 extraction API may be found in the following places:

- Geometry: section 1.2.3 and `C3GeoExt.c`.
- Texture: section 1.3.2 and `C3Texture.c`.
- Hierarchy: section 1.4.2 and `C3HieExt.c`.
- Animation: section 1.5.2 and `C3AnmExt.c`.

This appendix discusses how the C3 library internally extracts each of the preceding data.

E.2.4 C3OptimizeBeforeOutput

Once all of the relevant data has been loaded into the C3 library, the next stage is optimization using the function `C3OptimizeBeforeOutput` in `C3Out.h`. By calling this one function, data is processed in many ways:

- `C3TransformData`: Adjusts geometry, hierarchy, and animation data for hierarchy pivot points. See `C3CnvOpz.c`.
- `C3CompressData`: Removes duplicates and/or welds vertex attribute array. Also compresses texture paths for the TCS script file. See `C3CnvOpz.c`.
- `C3AssignVerticesToBones`: Error checks stitching or skinning information and converts vertex weights with bone names (`char*`) to a pointer to bone index (`u16*`). Also, ensures only one stitched or skinned mesh has been extracted. See `C3GeoCnv.c`.
- `C3ConvertActor`: Removes unused bones and animation tracks. Makes bone indices and control (transformation) indices sequential after hierarchy optimization. See `C3HieOut.c`.
- `C3SortWeightList`: For skinning, this function sorts the weights for each vertex according to final bone index to simplify processing. Also, for each vertex, this function removes any weight under 4% influence, and normalizes weights to 1. See `C3GeoCnv.c`.
- `C3ProcessOptionsAfterCompression`: Once final vertex attribute arrays are computed, this function calculates the optimal or user-defined quantization scale (or bit shift). See `C3Options.c`.
- `C3ConvertToStripFan`: If triangle-stripping option is enabled, this function transforms primitives to triangle strips and fans. See `C3CnvOpz.c`.

Even though `C3OptimizeBeforeOutput` is an optimization step, this function must be called after extraction because the next function, `C3WriteFile`, depends on it.

Each of these functions will be explained in more detail in subsequent sections.

E.2.5 C3WriteFile

After optimization, the next step is to convert and output all the data for the Character Pipeline. The function `C3WriteFile`, located in `C3Out.c`, simply calls these functions:

1. `C3WriteHierarchy` in `C3HieOut.c` that writes to ACT format.
2. `C3WriteGeometry` in `C3GeoOut.c` that creates display lists and converts/writes to GPL format.
3. `C3WriteAnimation` in `C3AnmOut.c` that writes to ANM format.

4. `C3WriteTextures` in `C3Texture.c` that write to TCS script and calls `TexConv.exe` to create the TPL format.
5. `C3WriteStatsFile` in `C3Stats.c` that writes geometry statistics and the information file.

These functions are largely format-dependent, but if you wish to write your own formats, you can replace these functions with your own conversion and output methods. All of the data is ready for output.

We will explain some of these functions in greater detail as we discuss each pipeline.

E.2.6 C3Clean

The last function that must be called is `C3Clean`. This function cleans up all allocated memory within the C3 library.

E.3 Geometry pipeline

Here is a step-by-step explanation of the path of geometry data in the C3 library.

E.3.1 Extraction

As we've mentioned, geometry data is extracted hierarchically in the order of object, primitive, and vertex. Here is a quick definition of terms for review:

- **Vertex:** defines attributes such as position, normal, texture coordinate, color, and weight.
- **Primitive:** a set of vertices defining triangles, quads, strips, fans, points, lines, and line strips.
- **Object:** a set of primitives controlled by some hierarchy transformation from the ACT format.

Because hierarchical data extraction is naturally stack-based and only one object, primitive, or vertex can be extracted at a time, an internal local data structure called `C3CurrentState` (defined in C3GeoExt.c) maintains this information:

```
// Defined in C3GeoExt.c
typedef struct
{
    C3GeomObject *geomObject; // currentObject
    C3Primitive  *primitive;  // currentPrimitive;
    C3Vertex     *vertex;     // currentVertex;
    C3PtrLink    *vertexPtr;  // currentVertexPtr;

} C3CurrentState;

// Local variable to C3GeoExt.c
static C3CurrentState   C3current;
```

Code 22 C3CurrentState

When `C3BeginObject` is called to create an object, object data is placed in `C3current.geomObject`. Then, when `C3BeginPolyPrimitive` or `C3BeginLinePrimitive` is called to create a triangle/quad or line, the appropriate primitive information is stored in `C3current.primitive`, and color and texture are inherited from the parent object. Likewise, calling `C3BeginVertex` places the vertex information in `C3current.vertex`. At the same time, it places a pointer to that vertex in `C3current.vertexPtr`, with color and texture inherited from its parent primitive. Primitives use the vertex pointer to refer to vertices (using the data structure `C3PtrLink`) in

order to promote the sharing of vertices among other primitives in the object, which aids in the creation of triangle strips and fans.

Calling any of the `C3End*` functions will insert the newly created data into the appropriate list. For objects, they are placed at the end of a local linked list, called `C3geomObjectList`, in `C3GeoExt.c`. Objects can be accessed later using some internal functions:

```
DSList* C3GetGeomObjList()
```

- **Purpose:** Returns the global linked list of geometry objects.
- **Argument:** None.

```
C3GeomObject* C3GetObjectFromIdentifier( char* identifier )
```

- **Purpose:** Returns the geometry object with the given identifier. `NULL` will be returned if no such named object exists.
- **Argument:** `identifier` should be the name of the geometry object when it was created with `C3BeginObject`.

```
C3GeomObject* C3GetNextObject( C3GeomObject* obj )
```

- **Purpose:** Returns the next geometry object after the given `obj`.
- **Argument:** `obj` should be any geometry object (should already be in the global `C3geomObjectList`).

Ending a polygon primitive using `C3EndPolyPrimitive` inserts the current primitive into the parent object. Depending on whether the number of vertices extracted was 3 or 4, the type will be assigned as a triangle or quad, respectively. If the source vertex order was counter-clockwise, the vertex order is then reversed to clockwise because the C3 library treats a clockwise vertex order as front-facing.

There are three variables in the structure `C3GeomObject` that maintain state about the object:

- `C3GeomObject.colorAllWhite` is a Boolean `C3Bool` that when true, indicates that all vertex colors for this object are white. If a texture is present and no normals are exported (i.e., lighting disabled at runtime), then the optimized blending operation is `GX_REPLACE` in the first TEV stage, rather than `GX_MODULATE`. This state is checked in the `C3SetColor` function.
- `C3GeomObject.useVertexAlpha` is another Boolean which is true when any vertex is using a non-opaque alpha (i.e., other than 255). At output, this variable helps in determining which quantization type (with or without alpha) to use for vertex colors for this particular object. This state is also checked in the `C3SetColor` function.
- `C3GeomObject.skinState` is an enumeration of `C3SkinState` that determines if the object is rigid (`C3_NONE`), if weights have been extracted (`C3_WEIGHTS_EXTRACTED`), if the object is correctly stitched (`C3_STITCHED`), or if the object is correctly skinned (`C3_SKINNED`). This state variable tells the C3 library how to process this object. This state is set in `C3SetWeight` and finalized with error checking in `C3AssignVerticesToBones`.

The `C3GeomObject` structure also holds minimum and maximum distance information for positions, texture coordinates, normals, and colors so that these vertex attributes can be put into hash tables. Using hash tables to compress (remove duplicates) and weld helps speed up the conversion time dramatically. The additional minimum and maximum information helps determine the optimal quantization shift (or power of two scale) later.

E.3.2 C3TransformObjectToPivot

The next stage in geometry data processing is `C3TransformObjectToPivot`. This function is called within `C3TransformData` in `C3CnvOpz.c`. It adjusts the vertex positions as a result of a rotation and scale pivot point that is different than the center position of the object's hierarchy transformation. This concept is better explained in the following diagram using a two-dimensional box as an example.

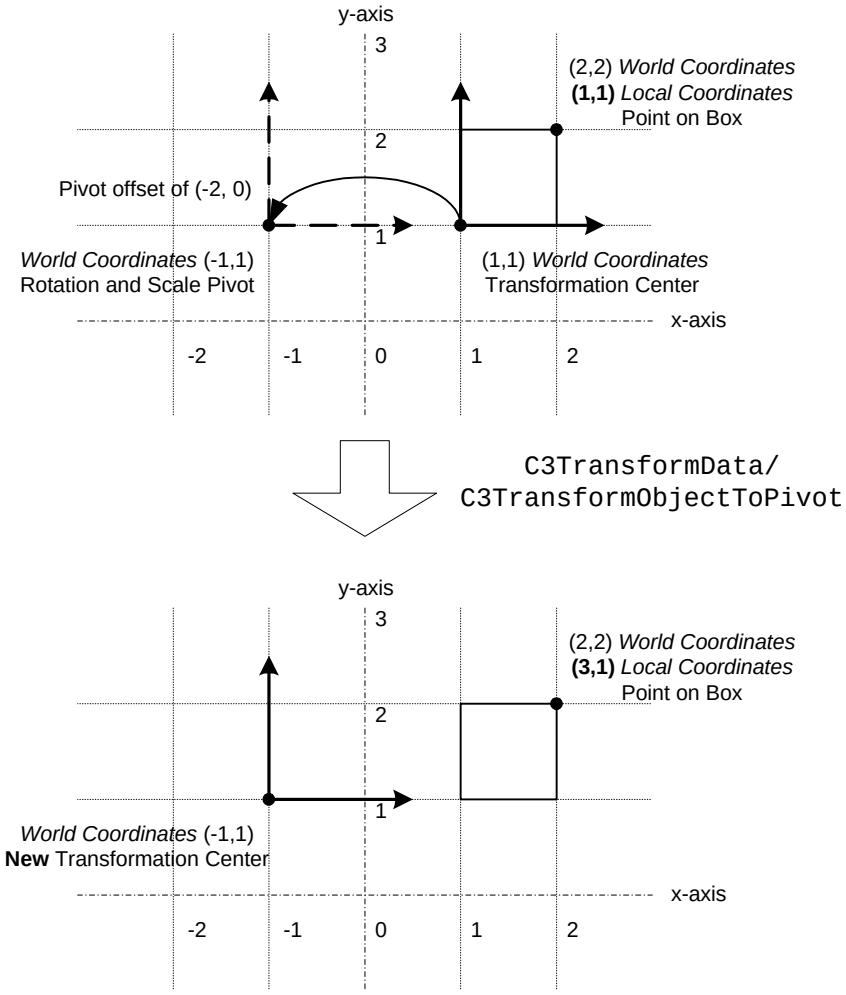

Figure 14 C3TransformObjectToPivot

Because the non-zero pivot offset of (-2, 0) forces a new center for the hierarchy transformation, the local coordinates of the box must be adjusted (bold type) so that the world coordinates of the box do not change. The pivot offset only translates the transformation center, which means the new transformation still has the same rotation and scale as the old transformation.

This step is only necessary when `C3SetPivotOffset` in the hierarchy is other than (0, 0, 0). This functionality in the C3 library is provided to support CG tools such as Maya, which allow the rotation and scale pivot position to be different than the center position of a hierarchy transformation. However, other CG tools, such as 3D Studio MAX, do not require this feature.

E.3.3 C3CompressData

After `C3TransformData`, the vertex positions are final. The next stage removes duplicates or weld elements in the vertex attribute arrays to take advantage of indexed display lists and to conserve memory. For each object, `C3CompressObjData` (located in `C3CnvOpz.c`) compresses or welds positions, color, texture coordinates, and normals using, respectively, `C3CompressPositions`, `C3CompressColor`, `C3CompressTexCoords`, and `C3CompressNormal`.

For conversion speed efficiency, vertex attributes are hashed into hash tables so that, on average, much fewer comparisons are needed to determine equality. Since the same basic method is used to compress or weld vertex attributes, let's focus on vertex positions to show how the C3 library performs compression and welding. Here is the code for `C3CompressPositions`:

```
static void
C3CompressPositions( C3GeomObject* obj )
{
    DSHashTable hTable;
    DSList      listArray[ FLT_TABLE_SIZE ]; // FLT_TABLE_SIZE is currently 1024
    C3Position  position;
    u16 unique    = 0;
    u16 i         = 0;
    u32 count     = 0;
    u16 numUnique = 0;

    if( obj->stats->numPositions <= 0 )
        return;

    // Initialize the hash table
    DSInitHTable( &hTable, FLT_TABLE_SIZE, listArray, HashPosition,
                  (Ptr)&position, &(position.link) );

    // Initialize the hashing parameters
    C3InitHashFloat( obj->minPosDistance, obj->maxPosDistance, FLT_TABLE_SIZE );

    // Hash the list
    C3HashList( &hTable, &obj->positionList );

    // Make each hash list "unique" by sorting by index
    for( i = 0; i < hTable.tableSize; i++ )
    {
        C3MakeIndexedListDataUnique( hTable.table + i, &unique, C3ComparePosition,
                                     numUnique, C3CopyPosition ); // numUnique is indexBase
        numUnique = numUnique + unique;
    }
    obj->stats->numUniquePositions = numUnique;

    if ( C3GetOptionWeldRadius( C3_TARGET_POSITION ) > C3_FLT_EPSILON )
    {
        // Weld positions (not just within the index)
        c3CurrentlyWelding = C3_TRUE;
        numUnique = 0;
        for( i = 0; i < hTable.tableSize; i++ )
        {
            C3MakeIndexedHTableDataUnique( &hTable, i, &unique, C3ComparePosition,
                                           numUnique, C3CopyPosition ); // numUnique is indexBase
            numUnique = numUnique + unique;
        }
        obj->stats->numPosWelded += obj->stats->numUniquePositions - numUnique;
        obj->stats->numUniquePositions = numUnique;
        c3CurrentlyWelding = C3_FALSE;
    }

    // Rebuild the list from the hash table
    DSHTableToList( &hTable, &obj->positionList );

    // Make each position in the list unique (remove positions with duplicate indices)
    C3CompressIndexedList( &obj->positionList );

    // Now, vertices can point to positions that are not in the positionList,
    // so we should repoint all of the vertex data to point within the list
    // for skinned objects.
    C3FixVertexPositionPtrs( obj );
}
```

Code 23 C3CompressPositions

During the extraction phase, the positions closest to and farthest from the origin, which were calculated previously, are stored as the hashing boundaries in `C3GeomObject.minPosDistance` and `C3GeomObject.maxPosDistance`. Currently, `FLT_TABLE_SIZE` is defined to be 1024, so there are 1024 lists, or buckets, into which positions can be hashed using the function `C3HashList`. This function works by taking each position in the linked list, calling `HashPosition` to find the hash index, and inserting the index into the hash table.

After all the positions are in the hash table, each index (or list) is traversed. `C3MakeIndexedListDataUnique` is invoked to compare the equality of vertex positions (using `C3ComparePosition`). Vertex positions with the same index are assumed to be equal. If the vertex positions are equal, then the function `C3CopyPosition` is called to make the two positions exactly the same. Indices are also generated in ascending order by using `numUnique` as the `indexBase` argument. This is very important because these indices will be used at runtime to reference the proper positions from the display list.

After compression, the welding phase is completed in a similar fashion using `C3MakeIndexedHTableDataUnique`, except that this function traverses each subsequent hash table index until no more welding within the welding radius can be done.

Finally, the hash table is converted back into a linked list structure using `DSHTableToList`. Since duplicates and welded positions have not been removed from the hash table, they will still exist in the linked list. Note that `C3CompressIndexedList` simply traverses the position linked list and removes all positions with duplicate indices (because positions with the same index can be assumed to be the same). This function does not deallocate memory, so use caution when using it. Positions do not need to be deallocated, since they utilize a memory pool and will be cleaned with `C3Clean` at the end.

A potential result of the removal of positions by `C3CompressIndexedList` is that the vertex position pointers may still be pointing to positions outside the object's position list (`C3GeomObject.positionList`). Normally this is not a problem; these positions have been copied by `C3CopyPosition` and are guaranteed to be exactly the same at output. However, you cannot traverse the position's link (`C3Position.link`) since it has been removed from the list. In skinning, this is unacceptable because all vertex position pointers must refer into the object's position list. Therefore, the function `C3FixVertexPositionPtrs` will traverse all of the vertex position pointers and force them to point within `C3GeomObject.positionList`.

Figure 15 illustrates the position compression algorithm. For simplicity, this figure does not contain any references to the hash table structure. The same process is followed for vertex normals, texture coordinates, and colors, but it does not complete the last step involving `C3FixVertexPositionPtrs`.

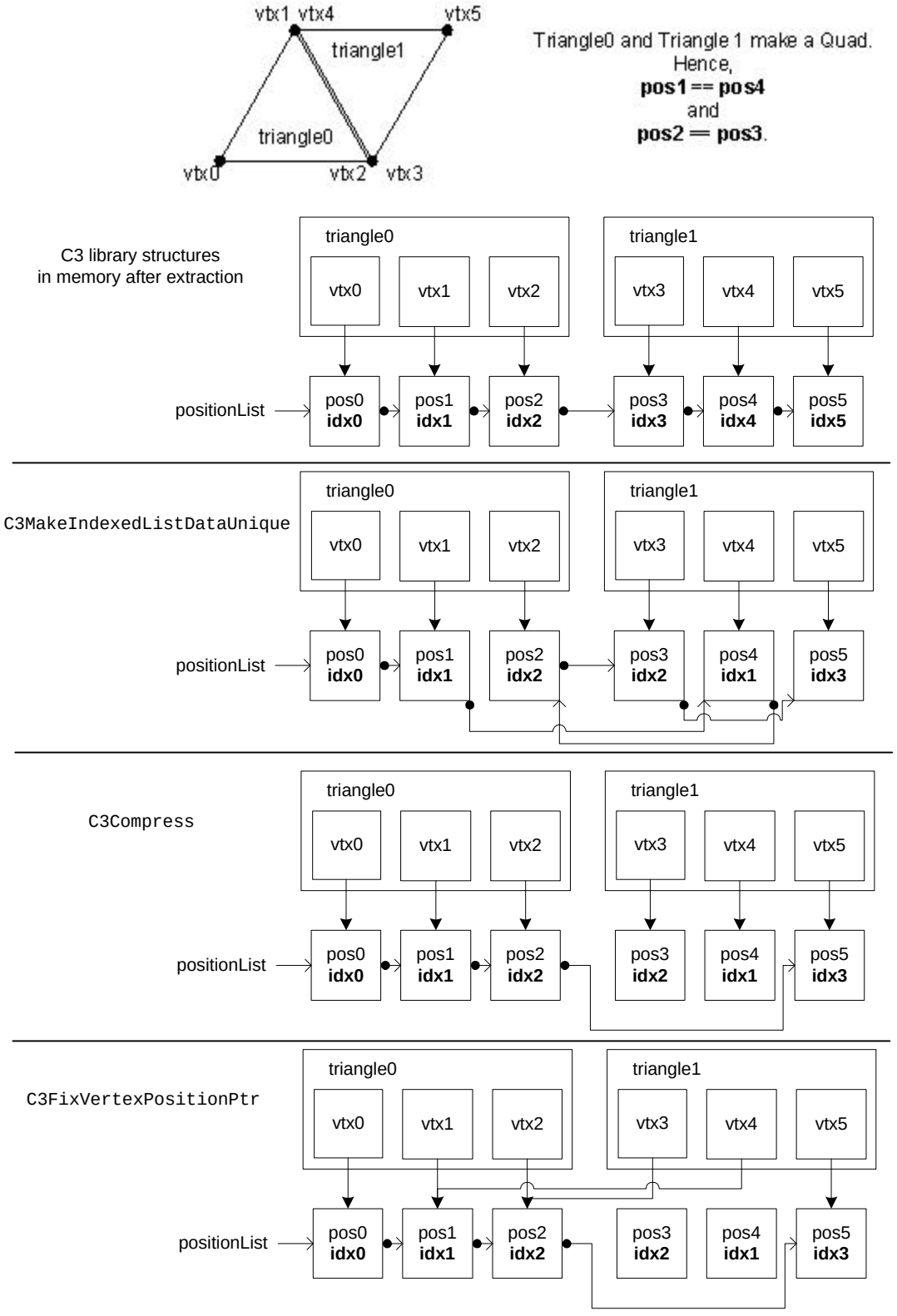

Figure 15 Compression of positions

E.3.4 C3AssignVerticesToBones

The main purpose of `C3AssignVerticesToBones`, which is located in `C3GeoCnv.c`, is to determine whether an object is rigid, stitched, or skinned. For review, here is a quick definition of terms:

- **Rigid:** All vertex positions in a geometry object are fixed and can only be modified as a group using the hierarchy transformation in the ACT file (rotation, scale, translation). The most common example is segmented characters.
- **Stitched:** Each vertex position in a geometry object is influenced by *one* hierarchy transformation, although transformations can be different for each of the vertices within a geometry object. Stitching is a subset of skinning where every vertex must have a weight of 100%.
- **Skinned:** Each vertex position in a geometry object can be influenced by a blend of multiple hierarchy transformations, depending on the weights assigned. The Character Pipeline implements CPU skinning rather than using pre-blended matrices in the graphics processor.

During extraction of vertex weights using `C3SetWeight`, vertices are attached to hierarchy nodes (or bones) using the name of the hierarchy node (`char* boneName`). Because of conversion efficiency, and because vertices refer to hierarchy transformations by index, `C3AssignVerticesToBones` converts each vertex's bone name to a pointer to the hierarchy node's index. We use a pointer because the hierarchy indices are not final until the bones have been pruned. Also, to prevent the removal of bones that have vertices attached to them, a Boolean `C3HierNode.usedForStitching` is set to `C3_TRUE`.

The C3 library calls `C3AssignVerticesToBones` only when weights have been extracted for a geometry object (i.e. `C3GeomObject.skinState == C3_WEIGHTS_EXTRACTED`). This function only works properly with this assumption.

The C3 library assumes a geometry object is assumed to be stitched until it encounters a vertex with more than one weight in the object. If C3 encounters a vertex with no weights, it cannot properly stitch or skin that geometry object. All weights are thus removed to make the object rigid.

E.3.5 C3SortWeightList

Once the geometry object is properly skinned (as checked by `C3AssignVerticesToBones`) and its bones have been pruned (by `C3ConvertActor`), then `C3SortWeightList` is a called to accomplish the following for every vertex:

- Sort the weight list by finalized bone index to simplify comparisons between weights.
- Make the bone indices unique among weights (e.g., if two weights, each 50%, are attached to the same bone index, then just make one weight that is 100%).
- Eliminate weights that have less than a 4% influence.
- Normalize weights to add up to 100%.

E.3.6 C3ProcessOptionsAfterCompression

Now all geometry data has been finalized and it is ready for format conversion and output. The quantization shift can be calculated, if the output quantization type is not a floating point number, in `C3ProcessOptionsAfterCompression`. The following quantization algorithm converts a floating point number to a 16-bit or 8-bit fixed point number:

1. Multiply a floating point number by the scale (or 2^{shift}).
2. Take the result and truncate the fractional component to create an integer number.
3. Cast the result to the appropriate output type (s16, u16, s8, u8).

Take care when selecting the scale. The integer number resulting from step 2 may not be appropriately represented when cast into the output type at step 3. For example, if the output type were u8, and the shift is 8 (meaning 2^8, or 256

scale), then true 1.0 cannot be represented. Instead, casting will clamp the value to 255. This means that when the value is dequantized, 255/256 (0.996) will result. Making the shift 7, or the scale 128, will allow 1.0 to be represented, but some precision will be lost.

Only the quantization type need be set in the options. The quantization scale is then calculated slightly differently for each vertex attribute (refer to section 1.2.4.3 for more information). In order to maximize fractional precision, however, all attributes use `C3ProcessOptionsAfterCompression` to calculate the largest absolute value before computing the quantization shift. The function `C3ComputeQuantizationShift` does the bulk of this work.

- For vertex positions, quantization is computed separately for skinned and non-skinned objects. Due to runtime constraints on our implementation of CPU skinning, vertex positions and normals in a skinned object are assumed to be quantized to a signed, 16-bit fixed point number. The largest absolute value of all positions in a skinned object is calculated separately from that of all positions in a non-skinned object; therefore, they can have separate scales.
- The C3 library normalizes vertex normals to be of length 1 at `C3SetNormal`, so the quantization scale is easy to compute, and set inside the call to `C3SetOptionQuantization` using the arguments `C3_TARGET_NORMAL` and channel 0.
- Quantization for texture coordinates is not stored globally in the structure `C3Options`; rather, it is stored per object in the `C3GeomObject.options` field (which is of a `C3OptionObj` type defined in `C3OptionsPrivate.h`).
- For keyframes, quaternions are always quantized to a s16 type with a shift of 14 because each *x*-, *y*-, *z*-, and *w*-coordinate will always be between 0.0 and 1.0. However, translation, scale, and Euler rotation components can be quantized to a different type. The quantization scale is computed optimally for these components per animation track and stored in `C3Track.paramQuantizeInfo`.

E.3.7 C3ConvertToStripFan

Next, the geometry pipeline converts triangles and quads to triangle strips and fans, if this option is enabled. We provide more information in section 1.2.4.2; the algorithm is outlined in Figure 4. The following code segment of `C3ConvertToStripFan` (in `C3CnvOpz.c`) controls how the extracted primitives are converted into strips and fans:

```
void
C3ConvertToStripFan( C3GeomObject* geomObj )
{
    C3CreateTriStripsGeomObject( geomObj );         // strip, tri
    C3ShortGeoStripsToQuads( geomObj );             // strip, tri, quad

    C3JoinGeoFans( geomObj );                       // strip, tri, quad, fan

    // Convert all quads to fans if C3_QUADS_TO_FANS is defined.
    // Non-coplanar quads are not a problem in the hardware, but there
    // can be backface rejection issues in Mac OpenGL, which the emulator uses.
#ifdef C3_QUADS_TO_FANS
    C3FansFromGeoQuads( geomObj );                  // strip, tri,       fan
#else
    C3ShortGeoFansToQuads( geomObj );               // strip, tri, quad, fan
#endif

    if( C3GetOptionEnableStripFanView() )
    {
        C3ColorVerticesByPrim( geomObj );
    }
}
```

Code 24 C3ConvertToStripFan

The algorithm is fairly simple and works well. Each function converts from one set of primitive types to another. At the end, primitives are colored according to their type if this option is enabled. However, if `C3SetOptionEnableStripFanView` is enabled, vertex colors will not be compressed because each polygon primitive will be flat-shaded (i.e., no Gouraud shading) at runtime. Both `C3SetOptionEnableStripFanView` and `C3SetOptionCompress` error-check this process.

The `C3_QUADS_TO_FANS` label, defined in `C3Defines.h`, converts all quads to fans with two triangles (as described in section 1.2.4.2). You should be aware of the `MAX_FAN_VTX` label defined at the top of `C3CnvOpz.c`. A bug in the first version of the graphics processor (HW1) necessitated this define to limit the number of vertices in a triangle fan to 4. If you are using the C3 library to convert to a later version of the graphics processor (HW2), set `MAX_FAN_VTX` to 255.

E.3.8 Conversion

The geometry data is now ready to be processed into the output GPL format and SKN format using two major functions. In `C3GeoCnv.c`, `C3ConvertToGeoPalette` is called once to convert each geometry object using `C3ConvertToDOLayout`. During the entire conversion phase, format and runtime data structures are allocated for everything in memory and filled in with the appropriate data. For example, the GPL header is a structure called `GEOPalette` and defined in `geoPalette.h`. It is allocated and set with the GPL version number, user data information, and information for each geometry object.

You may find it helpful to refer to "Game Engine Programming" (especially section 2.1) in this guide. Understanding how geometry data is used at runtime will help you to understand how and why geometry is converted in the C3 library.

Since CPU skinning was added to the Character Pipeline well after the initial design phase, the CP processes geometry information slightly differently between skinned objects and non-skinned objects. We will discuss the conversion of non-skinned objects first.

E.3.8.1 Non-skinned objects

The function `C3ConvertToDOLayout` converts one geometry object into GPL format. Each vertex attribute is converted using a separate function:

- `C3ConvertPositionData` converts positions for non-skinned objects.
- `C3ConvertLightingData` converts normals.
- `C3ConvertTextureData` converts texture coordinates.
- `C3ConvertColorData` converts colors.
- `C3ConvertDisplayData` creates display lists and state changes.

Conversion is straightforward and similar for all the vertex attributes, so as an example, let's explain how positions are converted. Code 25 shows the source code for `C3ConvertPositionData` with the code defined for `C3_GENERATE_NORMAL_TABLE` eliminated (this was a hack to aid in the creation of a normal table).

```
static void
C3ConvertPositionData( C3GeomObject* geomObj, DOPositionHeader** posHeader )
{
    DOPositionHeader* pos     = NULL;
    void*             array   = NULL;
    C3Position*       cursor  = NULL;
    u8                size    = 0;
    u32               count   = 0;
    FILE*             nFile   = NULL;

    C3_ASSERT( geomObj && posHeader && geomObj->positionList.Head );
```



```
    // Allocate a new header
    pos = (DOPositionHeader*)C3_CALLOC( 1, sizeof(DOPositionHeader) );
    C3_ASSERT( pos );

    pos->compCount = 3;
    pos->numPositions = geomObj->stats->numUniquePositions;
    pos->quantizeInfo = C3GetOptionQuantization( C3_TARGET_POSITION, 0 ); // we only look at type

    // Calculate the size of a position component
    size = C3GetComponentSize( C3_TARGET_POSITION, 0 );

    // Allocate the memory for the array
    pos->positionArray = array = C3_CALLOC( pos->numPositions,
                                            pos->compCount * size );
    C3_ASSERT( array );

    // Fill the array (Positions are assumed to have a unique index)
    cursor = (C3Position*)(geomObj->positionList.Head);
    while(cursor)
    {
        C3QuantizeFloat( array, pos->quantizeInfo, cursor->x, C3_FALSE );
        array = (void*)((u8*)array + size);
        C3QuantizeFloat( array, pos->quantizeInfo, cursor->y, C3_FALSE );
        array = (void*)((u8*)array + size);
        C3QuantizeFloat( array, pos->quantizeInfo, cursor->z, C3_FALSE );
        array = (void*)((u8*)array + size);
        count++;

        cursor = (C3Position*)(cursor->link.Next);
    }

    *posHeader = pos;

    C3_ASSERT( count == geomObj->stats->numUniquePositions );

    // Set the index quantization depending on the quant
    // NOTE: 0xFF or 0xFFFF index is reserved for positions by GX API
    if ( count <= 255 )
        C3SetVCDDataType( geomObj, C3_TARGET_POSITION, C3_VCD_INDEX8, 0 );
    else
        C3SetVCDDataType( geomObj, C3_TARGET_POSITION, C3_VCD_INDEX16, 0 );
}
```

Code 25 C3ConvertPositionData

Let's go over the algorithm:

1. First, a new header is allocated.
2. The header is filled with data which will be written out verbatim to the GPL file later:
   - The component count will always be 3, since vertex positions are always defined in 3D space: (*x*, *y*, *z*).
   - The number of positions is set from the statistics data structure.
   - Quantization information is set so the runtime code can properly dequantize position data.
   - The component size if calculated: 1 byte for u8 or s8, 2 bytes for u16 or s16, and 4 bytes for float. Multiplying the component count and size sets the proper stride in `GXSetArray` at runtime.
3. The array of positions is allocated.
4. Indices in the position list are in ascending order as completed by `C3CompressData`, so positions are written out in the proper quantization format using `C3QuantizeFloat`.
5. Finally, the optimal index size for the display list is selected depending on the number of positions. 8-bit indices are used if there are 255 positions or fewer.

All of the necessary information will be in memory and ready for output later.

E.3.8.2 Skinned objects

The GCN converts skinned objects in a slightly different fashion from non-skinned objects. Geometry data is specifically converted for use with the Character Pipeline's runtime CPU skinning library (see `/cp/build/libraries/skinning`). It may be difficult to understand the conversion algorithms without understanding the runtime algorithms. Therefore, we strongly recommend reading the runtime algorithm detailed in *NINTENDO GAMECUBE Development News* #2 (January 23, 2001). This article provides a critical understanding of how and why the C3 library converts skinning data in its current manner.

Due to runtime code optimizations, positions and normals must be interleaved in one array instead of two separate arrays. Moreover, both positions and normals must be quantized to a signed 16-bit fixed point number, using the same quantization scale. Since the Gekko CPU will transform vertices using a locked-cache DMA system, position/normal pairs cannot stride 32-byte cache line boundaries. Dummy vertices are therefore necessary in the position/normal array.

Since skinning was added to the Character Pipeline well after the initial design phase, all skinning data is gathered into a separate SKN format, which supplements the GPL format.

As you can see from the `C3ConvertToDOLayout` code, a different set of functions is called to convert positions and normals for if a skinned geometry object:

- `C3SortPositionNormalDataForSkin`.
- `C3ConvertSkinData`.
- `C3ConvertPositionNormalDataForSkin`.

Texture coordinates, vertex colors, and display data are still generated in the same manner as non-skinned objects.

`C3SortPositionNormalDataForSkin` checks for a one-to-one correspondence between each shared vertex and the positions and normals at that vertex. For example, it is possible that a vertex that is shared between two triangles in a strip has the same position, but two different normals. In this case, `C3SortPositionNormalDataForSkin` will create two position/normal pairs in order to properly transform the different normals on the CPU. The C3 library represents a position/normal pair by using the `C3Position` type to access the position data and then using the field `((C3Vertex*)C3Position.vtx)->normal` to access the normal data. At the end, this function will also do an initial sort on the linked list of positions (`C3GeomObject.positionList`), first by the number of weights, then by bone indices.

`C3ConvertSkinData` converts each position/normal pair into three lists depending on the number of weights: a one-matrix list (`SK1List`) for one bone influence, a two-matrix list (`SK2List`) for two bone influences, and the accumulation list (`SKAccList`) for three or more bone influences. Due to runtime constraints, each `SK1List` and `SK2List` must contain at least three position/normal pairs to be transformed. If there are less than three, then these weights are placed into the appropriate accumulation lists.

Normally, position/normal pairs with three or more bone influences would be distributed into the accumulation list. However, the function `C3SortAccListToTwoMtxList` performs an optimization at the end of `C3SortPositionNormalDataForSkin` that first sorts two of these weights into the two matrix lists, then sorts the leftovers into the accumulation list. At runtime, the two matrix lists are processed first, followed by each accumulation list. In this case, when the leftovers are finally accumulated on the CPU, they will be written somewhere into the `SK2List`, and as a result, this cache line must be flushed into main memory at runtime. `C3ConvertSkinData` computes exactly which indices in the position/normal pair array should be flushed; it is optimized using `C3CondenseFlushIndices` so that the same cache line is not flushed more than once.

`C3ConvertSkinData` also calculates the 32-byte cache line boundaries and inserts dummy, unused vertices so that position/normal pairs do not straddle a cache line. In the process, indices for the display list are properly updated. Due to limitations in the locked-cache DMA sizes, each `SK1List` is limited to 8192 bytes, and each `SK2List` and `SKAccList` is limited to 4096 bytes. At the end of this function, all of the organized data is written into the runtime skinning structures (defined in `SKNTypes.h`) so that it is ready for output in the SKN file format.

NOTE: For debug output, please uncomment the code at the end of `C3ConvertSkinData`. This file will show how all of the weights are distributed among the lists, along with a little bit of statistics.

E.3.8.3 Creating the display list

Creating the display list is the same for skinned and non-skinned objects. When display lists are created using `C3ConvertDisplayData`, the raw bytes in the GPL format are compatible with the graphics processor; therefore, `GXCallDisplayList` will work with just a pointer to it.

The current method of display list generation includes only geometry data, but no state changes, in the display list. The CPU sets state changes outside the display list via the GX library. Hence, geometry data is packaged into display lists that share the same runtime state. Figure 16 illustrates how state nodes and display lists are organized in the memory of the C3 library, and eventually into the GPL file.

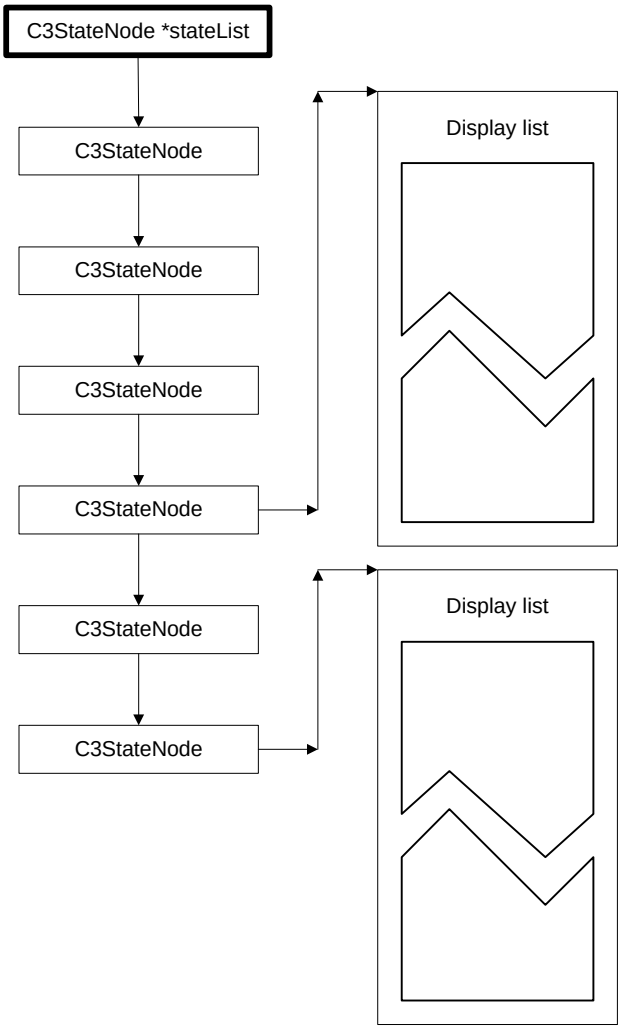

Figure 16 State nodes and display lists

The function `C3ConvertDisplayData` calls three functions:

- `C3SortPrimList`.

- `C3AssembleDispList`.
- `C3CreateDispList`.

First, minimizing state changes allows the graphics processor to draw more polygons. `C3SortPrimList` in `C3GeoCnv.c` performs an important task by sorting primitives in order of the most- to least-expensive state change. The following algorithm sorts the linked list of primitives in `C3GeomObject.primitiveList`:

1. All primitives are sorted by texture IDs using a quicksort algorithm with `C3ComparePrimForDispListOnTexture` as the comparison function.
2. If the object is stitched, primitives are sorted to minimize the number matrix loads. Here is the algorithm:
    - `C3SortPrimByPosMatrix` is called for each texture "bucket" from step 1.
    - `C3FindBestPrimitive`, going one primitive at a time, finds the primitive in the texture bucket that requires the least number of matrix loads, given the current set of matrices in the matrix cache. The best case is 0 (no loads).
    - If any matrices need to be loaded, matrices are replaced in a FIFO manner. The new set of matrices in the matrix cache is recorded in a `C3StateMatrixCache` data structure, which is then inserted into a linked list `matrixCacheList`. This list keeps a history of matrix caches.
    - After `C3SortPrimByPosMatrix` finishes processing every primitive in all the texture buckets, the function `C3GroupMatrixLoads` additionally optimizes hardware performance by loading as many matrices at a time as possible.
3. Primitives are then sorted by primitive type in the order of the `C3_PRIM_*` constants defined at the top of `C3GeoExt.h`.

Other state changes, such as vertex component descriptor (VCD) changes and TEV combine method, do not require sorting. They occur mostly when the texture state changes (by moving from a primitive with no texture to another with one texture, and vice versa), which has already been sorted. To understand how primitives are arranged before and after sorting, we highly recommend that you uncomment the "`TESTING`" code inside `C3SortPrimList`.

After sorting primitives, the next function called is `C3AssembleDispList`. This function steps through each primitive in `C3GeomObject.primitiveList` and creates the data structures shown in Figure 16. `C3GetStateChange` and `C3GetState` must figure out which states have changed between two primitives and store them in a `C3StateNode` data structure (defined at the top of `C3GeoCnv.c`) in the `stateList` linked list. The groups of primitives that share the same state are attached to the appropriate state node.

There are only four state changes that occur between display lists:

- `C3_STATE_TEXTURE0`: Sets the texture by changing the index into the TPL file. Also sets the wrapping mode and minification/magnification filter. Currently, only one texture channel is supported.
- `C3_STATE_TEV_COMBINE`: Sets the operation to be completed in the TEV stage using `GXTevOp`s.
- `C3_STATE_VCD`: Sets the vertex component descriptor (`GXAttrType`).
- `C3_STATE_MTXLOAD`: Loads a matrix derived from a control in the ACT file into hardware matrix memory.

After the data structure in Figure 16 is assembled, the last step is to invoke `C3CreateDispList`, which converts the structures in Figure 16 into GPL format. Specifically, `C3ConvertState` converts each state node and `C3ConvertPrimList` creates hardware-compatible indexed display lists in `C3GeoCnv.c`. The Character Pipeline runtime libraries process each state node into GX library state commands. If there is a valid pointer to a display list, `GXCallDisplayList` is invoked. Due to GX library specifications, each display list starts and ends at a 32-byte boundary (by padding, if necessary).

E.4 Hierarchy pipeline

The hierarchy pipeline is much simpler than the geometry pipeline.

E.4.1 Extraction

We recommend familiarizing yourself with the hierarchy extraction API covered in section 1.4.2. As hierarchy data is extracted using this API, information is stored in a `C3HierNode` data structure (defined in `C3HieExtPrivate.h`). Since the C3 library continues to use hierarchical extraction, the global variable `C3Stack` (declared at the top of `C3HieExt.c`) naturally uses a stack. The following local functions manipulate this stack in order to properly set parent information for hierarchy nodes:

- `void C3PushHierNode( C3HierNode *hNode )`.
- `void C3PopHierNode ( )`.

If calls to `C3BeginHierarchyNode` are not nested, the stack does not set parenting information automatically. Therefore, you will need to set parenting information explicitly using `C3SetParent`.

When a hierarchy node is completed using `C3EndHierarchyNode`, the `C3HierNode` structure is removed from the stack and inserted into a `DSTree` data structure. This tree structure is stored in the global variable `C3currentActor.hierarchy` (declared at the top of `C3HieExt.c`). The `DSTree` and `DSBranch` data structures from `Tree.h` implement a tree structure in a manner similar to how `DSList` and `DSLink` implement a flat linked list.

The transformation for a hierarchy node is stored in a `C3Control` structure (defined in `C3HieExtPrivate.h`). `C3Control` is a simple wrapper around the `CTRLControl` structure from the CTRL library. A control is the primary means for positioning, rotating, and scaling nodes in a hierarchy. In the Character Pipeline, the CTRL library is a simple and efficient method for constructing transformation matrices at runtime; the source code is referenced in `/cp/build/libraries/control`. For detailed information, please refer to section 2.5.2 in “Game Engine Programming” in this guide.

E.4.2 C3TransformBoneToPivot

If `C3SetPivotOffset` is used with a non-zero argument, the translation component in the hierarchy transformation control must be compensated. To understand this concept, please refer to section E.3.2 and Figure 14.

E.4.3 C3ConvertActor

After extraction, `C3ConvertActor` optimizes the hierarchy by calling `C3RemoveUnusedBones` (coded in `C3HieOut.c`) to remove unused hierarchy nodes. A bone is considered to be unused if the following three conditions exist:

- No geometry object is attached to the bone.
- No stitched or skinned vertices are attached to the bone.
- Both of the above conditions are true for all of this bone’s children.

As implied by the third point, these conditions are naturally checked. The recursive function `C3RemoveUnusedBones` traverses the hierarchy tree structure backwards from leaf nodes to parent nodes, checking for the first two conditions.

After all bones are pruned, bone and control indices are set in ascending order. All animation tracks that are attached to unused bones are discarded as well via `C3RemoveUnusedTracks`. To see the hierarchy tree before and after optimization, uncomment the `TESTING` code in `C3ConvertActor`.

E.4.4 Conversion and output

At this point, all hierarchy data is ready for output. Minimal conversion is necessary since control data and tree data are already stored in the runtime data structures `CTRLControl` and `DSTree`/`DSBranch`. Output occurs via `C3WriteHierarchy` (in `C3HieOut.c`). This function writes hierarchy data into the ACT format.

E.5 Animation pipeline

Processing animation in the Character Pipeline is straightforward, but a complete understanding requires quite a bit of math knowledge. We provide full source code in the Character Pipeline so that developers can see how animation data is converted from popular CG tool sources to an animation engine. Developers and artists can choose among the many different types of interpolation methods according to their game's requirements and tradeoffs.

Keep in mind that the current system is not game-specific. In the interest of readability, the code is not optimized in assembly language. Because the Character Pipeline animation system is only example code not designed for a game, we do not implement any solution for inverse kinematics (IK) or driven keys.

Please refer to section 2.4 in "Game Engine Programming," which outlines the architecture and runtime data structures used heavily by the C3 library.

E.5.1 Extraction

The C3 library API for extracting animation data is covered in section 1.5.2. For review, here is a quick definition of terms:

- **Keyframe:** A set of translation, rotation, and/or scale parameters that define a transformation at a moment in time.
- **Track:** A set of keyframes that define how a transformation control (i.e., one bone of a character) should animate according to time. In the C3 library, a bone can have only one track within one sequence.
- **Sequence:** A set of tracks that compose an animation for a character, such as walking, running, and jumping.

Like geometry and hierarchy data extraction, animation data is extracted in a hierarchical manner from sequence, to track, to keyframe. The current animation state is stored in a `C3AnimationState` structure allocated by `c3CurrentAnim` at the top of `C3AnmExt.h`. We recommend looking over all of the data structures in `C3AnmExtPrivate.h`.

Currently, only one sequence has been tested for export in the C3 library, so the same sequence name should be sent to all calls of `C3BeginTrack`. This is because 3D Studio MAX and Maya can only export one animation sequence per file. For details on how to use the AnmCombine tool to combine animation sequences after export, refer to Appendix D.

In extracting animation tracks, you should understand two flags in the `C3Track` structure:

- `sortKeyFramesNeeded`: If `C3_TRUE`, keyframes will have to be sorted so that their times are in ascending order. If `C3_FALSE`, keyframe times are already in the proper order for the ANM format.

- `replaceHierarchyCtrl`: In the current version of the C3 library, this flag is always `C3_TRUE`. This directs the Character Pipeline runtime library to ignore the hierarchy transformation control (matrix), and to use only the results of keyframe interpolation to determine the final transformation for a bone.

  NOTE: This flag exists to correct an inefficiency in a previous version of the Character Pipeline. Previously, animation keyframes were converted to be relative to the hierarchy transformation. The final transformation for a bone involved a runtime matrix concatenation of the animation matrix and the orientation (hierarchy) matrix. This turned out to be unnecessary, since animation data from CG tools are not relative to the hierarchy. However,

this flag is necessary for backward compatibility so that the runtime library can still properly animate older ANM files.

When a track is completed with `C3EndTrack`, the CP inserts it as a `C3AnimBank` structure into a linked list of tracks in the global animation bank `c3CurrentAnim.bank`. The track is also inserted into a linked list of `C3SeqTrack` structures inside the sequence at `C3Sequence.trackList`. When a keyframe is completed with `C3EndKeyFrame`, it is inserted into a `C3Track.keyList` structure as a linked list of keyframes within its parent track.

After the extraction phase, the function `C3TransformData` prepares all animation data for export by calling four major functions:

- `C3SortKeyFrames`.
- `C3TransformTrackToPivot`.
- `C3ComputeTrackBezierInOutControl`.
- `C3ComputeTrackInOutControl`.

E.5.2 C3SortKeyFrames

As previously explained, if the `C3Track.sortKeyFramesNeeded` flag is `TRUE` for a track, `then C3SortKeyFrames` will sort the keyframes in ascending time order.

E.5.3 C3TransformTrackToPivot

Just the use of `C3SetPivotOffset` compensates the position of geometry vertices and the translation of hierarchy controls, `C3TransformTrackToPivot` must be called to compensate the translation component of the animation track. These adjustments maintain flexibility for artists when using Maya. Animation data is transformed to hierarchy data in a similar fashion (see section E.3.2 and Figure 14).

E.5.4 C3ComputeTrackBezierInOutControl

`C3ComputeTrackBezierInOutControl` computes the control points for Bezier-interpolated components.

At extraction, the *in* and *out* tangent information is stored as an angle (in radians) to determine the shape of the interpolation curve; however, the Character Pipeline runtime libraries need Bezier control points. Fortunately, control points are easy to compute, given the angle:

1. First, find the *in* and *out* tangent control points between keyframes *k0* and *k1*. The horizontal *u*-axis is time, and the vertical *v*-axis is the value.
2. Compute the time difference as *du* = time(*k1*) – time(*k0*).
3. The first control point (*out* tangent of *k0*) lies 1/3 of the time distance from *k0* to *k1*. So, using simple trigonometry, *outtan*(*k0*) = *outtan_angle*(*k0*) * *du*/3.
4. The second control point (*in* tangent of *k1*) lies 1/3 of the time distance from *k1* to *k0*. So, *intan*(*k1*) = *intan_angle*(*k1*) * *du*/3.

This process must be repeated for each (*x*, *y*, *z*) for each Bezier-interpolated translation, Euler rotation, and scale component.

E.5.5 C3ComputeTrackInOutControl

This function will compute the *in* and *out* controls necessary for three types of interpolation:

- **Squad:** quadratic interpolation of quaternions for rotation
- **SquadEE:** quadratic interpolation of quaternions with ease and TCB (tension, continuity, bias) parameters for rotation.
- **Hermite:** Hermite-based interpolation using TCB parameters for translation and scale.

For squad-type interpolations, you must call `C3MakeTrackQuaternionsClosest`. In quaternion math, two "opposite" quaternions can equally represent one rotation. Therefore, given a quaternion *q0*, to interpolate to the next quaternion *q1* properly, we need to find the "closest" quaternion *q1* to *q0*.

For squad interpolation, getting the *in* and *out* controls is simple in `C3ComputeTrackSquadAB`: the *in* and *out* control for a keyframe is simply the quaternion of the previous and the next keyframe, respectively.

At this point, we'd prefer to spare the reader any more discussion of difficult, four-dimensional quaternion mathematics. The algorithm for computing tangent controls for squad interpolation with ease and TCB parameters, accomplished in `C3ComputeTrackSquadTCB`, is rather hard to explain. The algorithm for the proper *in* and *out* controls needed for TCB interpolation for translation and scale animation, performed in `C3ComputeTrackTCB`, is equally abstruse. We implement these esoteric interpolation methods in order to faithfully represent 3D Studio MAX animation and give more flexibility to artists and animators. Although these methods may not be helpful for a realistic game engine, example code exists for your perusal nonetheless.

E.5.6 Conversion and output

Once all of the necessary *in* and *out* controls have been computed, all of the animation data is ready for output.

`C3CreateAnimBank` starts the conversion. Since the C3 library stores animation data in structures similar to the runtime animation structures, conversion is simple. In functions such as `C3ConvertTracks` and `C3ConvertKeyFrames`, animation data is just transferred from C3 library structures (i.e., `C3Track` and `C3KeyFrame`) to runtime structures (i.e., `ANIMTrack` and `ANIMKeyFrame`).

Afterward, `C3WriteAnimBankInt` writes out the structures to the ANM file format. Quaternions are always quantized to a signed 16-bit fixed point number because each *x*, *y*, *z*, and *w* component is always between 0 and 1. In the C3 library, translation and scale components can be quantized to a user-defined type; however, runtime interpolation will be slower due to unoptimized animation code.

E.6 Texture pipeline

Since the TC library is separate from the C3 library, the texture pipeline creates a TCS script file for use by the texture converter application `TexConv.exe`. All of the functions are contained within `C3Texture.h`, `C3TexturePrivate.h`, and `C3Texture.c`.

E.6.1 Extraction

The API for setting texture data is explained in section 1.3.2. For review, the following list defines the four major components in a TCS script file:

- **Source (file):** A file name that is a location for a color map, alpha map, or palette.
- **Image:** Two source files that combine a color and alpha map. An alpha map is not necessary.
- **Palette:** One source file that specifies a palette (CLUT) for a color-indexed (CI) texture.
- **Texture:** A binding between an image and a palette. A palette is not necessary.

Information for each of these four groups is extracted by the following functions. These functions fill separate data structures and store them in separate linked lists allocated at the top of `C3Texture.c`.

- `C3BeginTexture`/`C3EndTexture` functions fill a `C3Texture` structure and add it to the `c3TexList` list.

- `C3SetImage` fills a `C3Image` structure for a color map and adds it to the `c3ImgList` list.
- `C3SetImageAlpha` fills a `C3Image` structure for an alpha map and adds it to the `c3ImgList` list.
- `C3SetPalImage` fills a `C3Palette` structure and adds it to the `c3PalList` list.

Since a source filename is necessary for the latter three functions, they also call `C3AddSrcImg` to fill a `C3SrcImage` structure and add it to the `c3SrcList` list.

As you know, texture data is assigned for a geometry object or a primitive depending on the context in which `C3BeginTexture` is called. It is possible to assign a texture to a geometry object, then override it with a different texture on a per-primitive basis.

Texture formats are set based on a current state stored in a `c3TexFmt` variable at the top of `C3Texture.c`. Using `C3SetTexFmt` changes the state for the output texture format. For example, the current texture converter only supports color-indexed textures in the `CI8` format and palettes in the `RGB565` format, so a color-indexed texture must be extracted in the following manner in the C3 library:

```
C3BeginTexture( 0 );
C3SetTexFmt( CI8 );
C3SetImage( “C:\\sometex.tga” );
C3SetTexFmt( RGB565 );
C3SetPalImage( “C:\\sometex.tga” )
C3EndTexture();
```

Code 26 Extracting color-indexed textures

If the second call to `C3SetTexFmt` did not exist, then the palette output format would still be `CI8` and the texture converter would produce an error. Every time `C3BeginTexture` is called, the format defaults back to `RGBA8`, so we recommend calling `C3SetTexFmt` with the desired format before calling `C3SetImage`, `C3SetImageAlpha`, or `C3SetPalImage`.

In Code 26, `C3SetTexTiling` and `C3SetTexFilterType` are not called; therefore, default settings are used. The default tiling clamps both *s*- and *t*-axes, and the default filter type uses point samples.

E.6.2 C3CompressTextureData

After extracting all the texture data, the only optimization step is to remove duplicate information in `C3CompressTextureData` (called within `C3CompressData`). `C3CompressTextureData` calls `C3CompressListData` on each of the four linked lists, using a hash table to speed compression time. This method of compression is a similar but simpler version of how geometry vertex attributes are compressed in section E.3.3.

E.6.3 Output

No conversion is necessary, since all of the texture data is naturally processed into the four groups for the TCS script file. The function `C3WriteTextures` first calls `C3OutputTexScript` to write the TCS script file, then it invokes the texture converter application to create the TPL file.

At output, each of the four lists are written out using the following functions:

- `C3OutputSrcImg` writes out the “`file n =`” lines, where `n` is an integer.
- `C3OutputImg` writes out the “`image n =`” lines.
- `C3OutputPal` writes out the “`palette n =`” lines.
- `C3OutputTex` writes out the “`texture n =`” lines.

